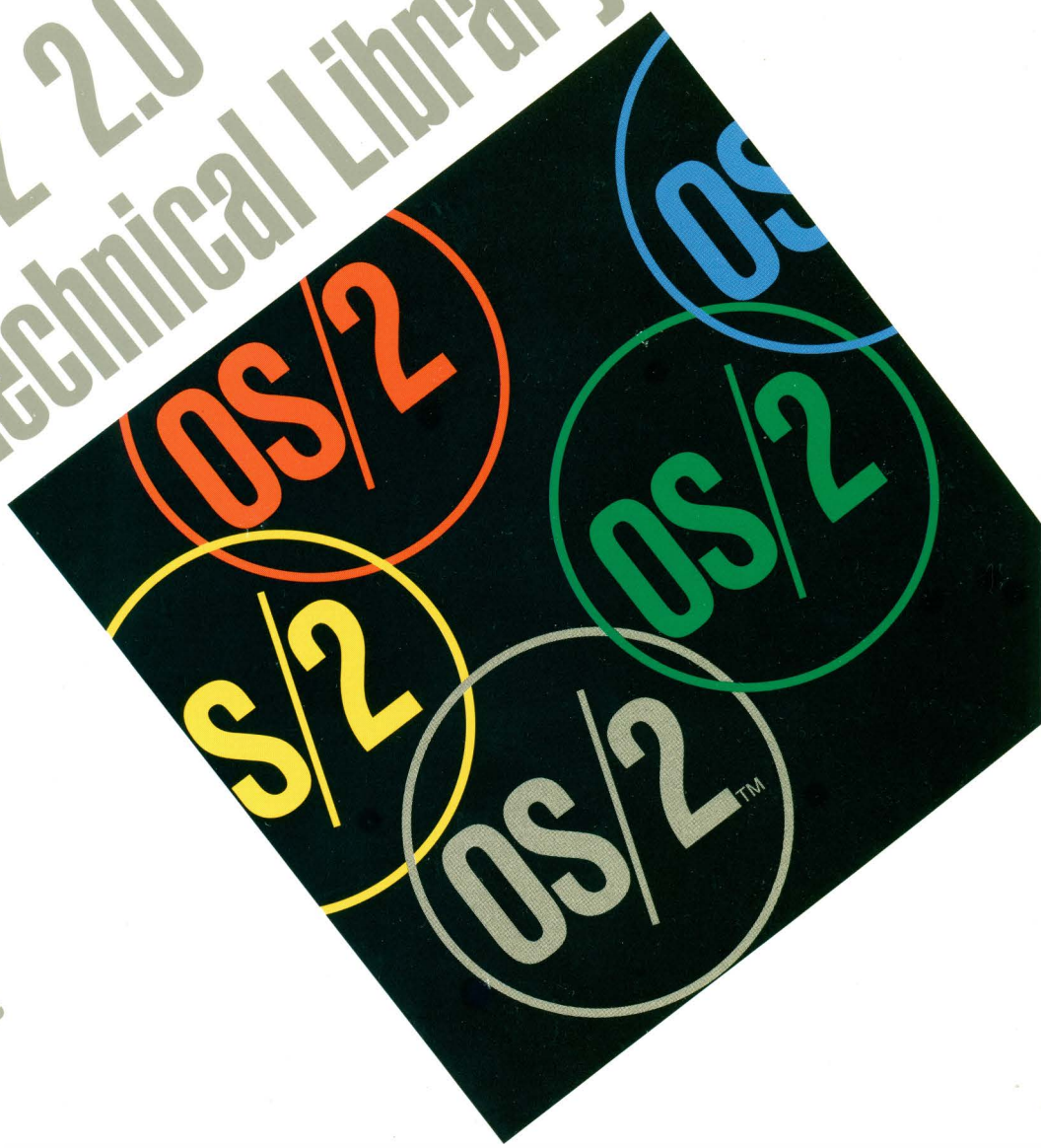
# OS/2 2.0 Technical Library




## Application Design Guide

Version 2.00

IBM Programming Family

# OS/2 2.0 Technical Library

## Application Design Guide

Version 2.00



Programming Family

**Note**

Before using this information and the product it supports, be sure to read the general information under "Notices" on page xv.

**First Edition (March 1992)**

# Contents


**Notices** . . . . . xv
Trademarks and Service Marks . . . . . xv
Double-Byte Character Set (DBCS) . . . . . xvi

**About This Book** . . . . . xvii
Who Should Read This Book . . . . . xvii
How This Book is Organized . . . . . xvii
OS/2 2.0 Technical Library . . . . . xviii

**Chapter 1. OS/2 2.0 Overview** . . . . . 1-1
OS/2 2.0 Highlights . . . . . 1-1
386 Features . . . . . 1-1
Portability . . . . . 1-1
Compatibility with Version 1.X . . . . . 1-2
Multiple DOS Sessions . . . . . 1-2
Virtual Device Drivers . . . . . 1-2
The OS/2 Operating System and Presentation Manager Program . . . . . 1-2
Queued Input . . . . . 1-3
Device-Independent Graphics . . . . . 1-4
Shared Resources . . . . . 1-4
Control Program Fundamentals . . . . . 1-4
Multitasking . . . . . 1-4
Sessions . . . . . 1-5
Processes . . . . . 1-6
Threads . . . . . 1-6
Dynamic Linking . . . . . 1-7
Memory Management . . . . . 1-7
The File System . . . . . 1-8
Interprocess Communication . . . . . 1-10
Semaphores . . . . . 1-10
Pipes . . . . . 1-10
Queues . . . . . 1-11
Shared Memory . . . . . 1-11
Exception Handling . . . . . 1-11
Multiple DOS Sessions . . . . . 1-11
Device Support . . . . . 1-12
PM Fundamentals . . . . . 1-13
The Window Environment . . . . . 1-13
Defining Window Relationships . . . . . 1-13
Creating and Classifying Windows . . . . . 1-15
Providing the User Interface . . . . . 1-16
Standard and Control Windows . . . . . 1-16
Primary and Secondary Windows . . . . . 1-18
Dialog Box . . . . . 1-18
Handling Mouse and Keyboard Input . . . . . 1-19
Processing Messages . . . . . 1-19
Handling Application Resources . . . . . 1-21
Resource Editors . . . . . 1-22
Exchanging Data Among Applications . . . . . 1-22
User-Generated Data Exchange . . . . . 1-22
Application-Generated Data Exchange . . . . . 1-23
Direct Manipulation . . . . . 1-23

Information Presentation Facility . . . . . 1-23
Coding the Application . . . . . 1-24
Developing the Help Information . . . . . 1-24
Presentation Drivers . . . . . 1-24
The Graphics Programming Interface . . . . . 1-24
Presentation Spaces and Device Contexts . . . . . 1-24
Graphics Primitives . . . . . 1-25
Graphics Objects and Operations . . . . . 1-26
Path . . . . . 1-26
Bit Map . . . . . 1-26
Font . . . . . 1-26
Logical Color Palette . . . . . 1-26
Clipping . . . . . 1-27
Transformation . . . . . 1-27
Drawing . . . . . 1-27
Retained Graphics and Segments . . . . . 1-28
Metafiles . . . . . 1-28
Producing Hard-Copy Output . . . . . 1-28
The OS/2 Application Programming Interface Functions . . . . . 1-29

**Chapter 2. The 32-bit OS/2 Programming Environment** . . . . . 2-1
Intel 80386 Architecture . . . . . 2-1
Physical Characteristics . . . . . 2-2
Memory Addressing . . . . . 2-3
Real Mode . . . . . 2-3
Protect Mode (Segmented Memory Model) . . . . . 2-4
Protect Mode (Flat Memory Model) . . . . . 2-6
Paging . . . . . 2-6
Protection . . . . . 2-8
Type Checking . . . . . 2-8
Limit Checking . . . . . 2-8
Privilege Levels . . . . . 2-9
Restriction of Procedure Entry Points . . . . . 2-9
Reserved Instructions . . . . . 2-10
Interrupts . . . . . 2-10
Input/Output Processing . . . . . 2-11
Virtual 8086 Mode . . . . . 2-11
Numeric Coprocessor . . . . . 2-12
Coprocessing . . . . . 2-13
OS/2 and the 80386 Processor . . . . . 2-13
Process Address Space . . . . . 2-14
Memory Objects and Memory Sharing . . . . . 2-15
Page Attributes and Memory Access Protection . . . . . 2-17
Compatibility with 16-Bit OS/2 . . . . . 2-18
Summary . . . . . 2-20

**Chapter 3. The Application Development Environment** . . . . . 3-1
Applications Running Under OS/2 . . . . . 3-1
Full-Screen Applications . . . . . 3-2
Windowable Applications . . . . . 3-2
PM Applications . . . . . 3-3
DOS/Windows Applications . . . . . 3-4
Programming Models . . . . . 3-5
Pure 16-Bit Applications . . . . . 3-5
Mixed 16-Bit Applications . . . . . 3-6
Pure 32-Bit Applications . . . . . 3-7

Mixed 32-Bit Applications . . . 3-8
The Program Development Environment . . . 3-9
Include File Architecture . . . 3-10
C Compiler Support . . . 3-11
Library Support . . . 3-12
Mixing 16-Bit and 32-Bit Code . . . 3-12
Thunking . . . 3-12
32-Bit OS/2 Memory Layout . . . 3-15
Flat Memory . . . 3-15
Tiled Memory . . . 3-15
Different Parameter Sizes . . . 3-18
64K Segment Boundary Problems . . . 3-18
Different Call Models . . . 3-19
Calling 16-Bit Code from 32-Bit Code . . . 3-19
Using the _Seg16 and _Far16 _Pascal keywords . . . 3-20
Formal Parameters . . . 3-21
Examples of using _Far16 _Pascal and _Seg16 . . . 3-25
Function Calls to 16-Bit Modules . . . 3-29
Using 16-Bit Window Procedures . . . 3-29
Creating a Window . . . 3-29
Passing Messages to 16-Bit Windows . . . 3-30
Passing Messages to 32-Bit Windows . . . 3-31
Calling 32-Bit Code from 16-Bit Code . . . 3-33
Migrating to OS/2 2.0 . . . 3-33
Summary . . . 3-35

**Chapter 4. Comparison of 16-Bit and 32-Bit OS/2 Functions** . . . 4-1
Changes to the Control Program . . . 4-1
Memory Management . . . 4-1
Allocating Memory . . . 4-2
Freeing Memory . . . 4-4
Suballocating Memory . . . 4-4
Using Named Shared Memory . . . 4-6
Using Unnamed Shared Memory . . . 4-8
Generating Dynamic Code . . . 4-9
Determining Available Memory . . . 4-11
Discarding Memory Objects . . . 4-11
Setting Memory Commitment and Access . . . 4-11
Checking a Process's Virtual-Memory Map . . . 4-11
Threads and Processes . . . 4-12
Creating Threads . . . 4-12
Controlling Threads . . . 4-13
Exiting from Threads and Processes . . . 4-13
Ending Other Processes . . . 4-14
Handling Critical Sections . . . 4-14
Waiting for Threads . . . 4-14
Getting Thread and Process Information . . . 4-14
Starting Programs . . . 4-16
Debugging Programs . . . 4-16
16-Bit Functions with No 32-Bit Counterparts . . . 4-17
Semaphores . . . 4-17
Using Semaphores . . . 4-18
Signalling Events with Semaphores . . . 4-20
Using Event Semaphores Between 16- and 32-bit Code . . . 4-21
Using Semaphores for Mutual Exclusion . . . 4-21
Using Semaphores for Multiple Waiting . . . 4-25

Unnamed Pipes . . . . . 4-26
Named Pipes . . . . . 4-27
Queues . . . . . 4-27
Timers . . . . . 4-28
Dynamic Linking . . . . . 4-28
Device I/O . . . . . 4-30
File Systems . . . . . 4-30
Searching Directories . . . . . 4-32
Querying File Mode . . . . . 4-33
Querying System Information . . . . . 4-33
Reading Asynchronously . . . . . 4-33
Setting the File Mode . . . . . 4-33
Setting Available Number of File Handles . . . . . 4-34
Writing Asynchronously . . . . . 4-34
Message Retrieval . . . . . 4-34
Code-Page Management . . . . . 4-34
Session Management . . . . . 4-35
Error Management . . . . . 4-35
Signals . . . . . 4-35
Exception Management . . . . . 4-36
VDD Services . . . . . 4-37
Support for 16-Bit Subsystems . . . . . 4-37
Changes to Presentation Manager Services . . . . . 4-37
Printing . . . . . 4-38
Workplace . . . . . 4-39
Customizing Help Information . . . . . 4-40
32-Bit Migration . . . . . 4-41
Standard Font-and File-Dialog Boxes . . . . . 4-42
Window Controls . . . . . 4-44
Notebook . . . . . 4-44
Container . . . . . 4-45
Value Set . . . . . 4-48
Slider . . . . . 4-49
Pop-Up Menus . . . . . 4-51
Desktop Background . . . . . 4-51
Hooks . . . . . 4-51
Paths, Regions, and Bit Maps . . . . . 4-51
Fonts and Characters . . . . . 4-51
Polylines . . . . . 4-52
Transformations . . . . . 4-52
PM Helper Macros . . . . . 4-52
Summary . . . . . 4-54

**Chapter 5. Dynamic Linking** . . . . . 5-1
Static vs. Dynamic Linking . . . . . 5-1
Load-Time Dynamic Linking . . . . . 5-3
Run-Time Dynamic Linking . . . . . 5-6
DLL Data . . . . . 5-7
DLL Initialization and Termination . . . . . 5-9
Building DLLs . . . . . 5-9
External Function References . . . . . 5-10
Module-Definition Files . . . . . 5-10
Import Libraries . . . . . 5-10
Creating a Simple DLL . . . . . 5-11
Importing DLL Functions . . . . . 5-12
Using an Import Library . . . . . 5-12

Using Shared and Instance Data . . . . . 5-13
Creating an Initialization/Termination Function . . . . . 5-13
Linking at Run Time . . . . . 5-18
Protected Memory Use . . . . . 5-19
DLL Side Effects . . . . . 5-22
Summary . . . . . 5-23

**Chapter 6. Multiple Virtual DOS Sessions** . . . . . 6-1
Overview . . . . . 6-1
Enhanced DOS Sessions . . . . . 6-1
Fast Mode Switching . . . . . 6-3
Multiple DOS Sessions . . . . . 6-5
DOS Settings . . . . . 6-5
Transfer of Data Between DOS Sessions . . . . . 6-6
Increased Available Memory . . . . . 6-6
Memory Extender Support . . . . . 6-6
Expanded Memory Specification . . . . . 6-6
Extended Memory Specification . . . . . 6-8
DOS Protect Mode Interface . . . . . 6-10
Inside Enhanced DOS Session . . . . . 6-13
Virtual Device Helper Services . . . . . 6-13
The Virtual Device Driver Model . . . . . 6-14
Communication with OS/2 Processes . . . . . 6-17
Summary . . . . . 6-18

**Chapter 7. Object-Oriented Programming Using SOM** . . . . . 7-1
Object-Oriented Programming . . . . . 7-1
Object-Oriented Programming Example . . . . . 7-1
IBM System Object Model . . . . . 7-3
SOM Features . . . . . 7-3
Encapsulation . . . . . 7-3
Inheritance . . . . . 7-4
Polymorphism . . . . . 7-4
The SOM Run-Time Environment . . . . . 7-4
Creating SOM Classes . . . . . 7-6
Object Interface Definition Language . . . . . 7-6
Processing Class Definition Files . . . . . 7-11
A Simple Class Implementation . . . . . 7-13
SOM Macros, Functions, and Data . . . . . 7-16
Class-Specific SOM Macros . . . . . 7-16
General SOM Macros and Functions . . . . . 7-18
Invoking Methods and Accessing Data . . . . . 7-20
A SOM Client Program . . . . . 7-23
Inheritance and Polymorphism: Overriding Methods . . . . . 7-24
Metaclasses . . . . . 7-28
Implied Metaclasses . . . . . 7-30
Building SOM Class Libraries . . . . . 7-32
SOM ANIMALS Sample Program in the OS/2 2.0 Toolkit . . . . . 7-33
SOM ANIMALS Sample Program with Implied Metaclasses . . . . . 7-33
Summary . . . . . 7-46

**Chapter 8. Workplace Programming Interface** . . . . . 8-1
CUA Guidelines for an Object-Oriented User Interface . . . . . 8-1
Objects, Classes, Hierarchies, and Inheritance . . . . . 8-2
Views of Objects . . . . . 8-2
Classes of Objects . . . . . 8-3

Object Relationships . . . . . 8-3
Interaction with Objects . . . . . 8-4
Designing an Object-Oriented User Interface . . . . . 8-4
Defining the Objects for a Software Model . . . . . 8-5
Determining Object Relationships and Behaviors . . . . . 8-5
Determining the Necessary Views . . . . . 8-6
Determining the Action Choices . . . . . 8-6
The OS/2 Object-Oriented User Interface: The Workplace Shell . . . . . 8-6
The OS/2 2.0 Workplace Programming Interface . . . . . 8-8
Designing Workplace Classes . . . . . 8-9
Settings-Notebook Methods . . . . . 8-10
Pop-Up Menus . . . . . 8-12
Object Information Methods . . . . . 8-22
Direct Manipulation Methods . . . . . 8-28
Save/Restore State Methods . . . . . 8-29
Object Usage Methods . . . . . 8-30
Setup/Cleanup Methods . . . . . 8-32
Workplace Class Methods: Implied Metaclasses . . . . . 8-35
Creating a Workplace Object: The Car Object . . . . . 8-37
The Workplace Application Interface . . . . . 8-39
Object Class Functions . . . . . 8-40
Object Instance Functions . . . . . 8-41
REXX Utility Workplace Functions . . . . . 8-42
Installing a Workplace Object . . . . . 8-42
Object Installation Programs . . . . . 8-42
Object Installation Batch Files . . . . . 8-44
The Workplace Class List Object . . . . . 8-45
Programming Considerations for the Workplace . . . . . 8-46
Summary . . . . . 8-48

**Appendix A. Sample Programs Cross Index** . . . . . A-1
Control Program Functions . . . . . A-1
Device Functions . . . . . A-5
Direct Manipulation Functions . . . . . A-6
GPI Functions by Functional Area . . . . . A-7
Profile Functions . . . . . A-11
Window Functions by Functional Area . . . . . A-12

**Index** . . . . . X-1

# Figures


1-1. Multitasking Hierarchy . . . . . 1-5
1-2. Windows on the Screen . . . . . 1-14
1-3. Window Hierarchy . . . . . 1-14
1-4. Standard Window . . . . . 1-17
1-5. Message-processing System . . . . . 1-20
2-1. 80386 General, Segment and Status Registers . . . . . 2-2
2-2. Real Mode Addressing . . . . . 2-4
2-3. Protect Mode Addressing—Without Paging . . . . . 2-5
2-4. Protect Mode Addressing—With Paging . . . . . 2-7
2-5. Virtual 8086 Environment—Memory Management . . . . . 2-12
2-6. OS/2 2.0 Process Address Space . . . . . 2-14
2-7. Multiple Linear Address Space Management . . . . . 2-16
2-8. Tiled Address Space Next to LDT . . . . . 2-18
3-1. OS/2 2.0 Application Types . . . . . 3-1
3-2. PM Application Template . . . . . 3-3
3-3. Building a Pure 16-Bit Application . . . . . 3-6
3-4. Building a Mixed 16-Bit Application . . . . . 3-7
3-5. Building a Pure 32-Bit Application . . . . . 3-8
3-6. Building a Mixed 32-Bit Application . . . . . 3-9
3-7. Template for the OS2.H Include File . . . . . 3-10
3-8. Stack frame for VioXXX Example . . . . . 3-12
3-9. 16 to 32-Bit Application/Subsystem Interactions . . . . . 3-13
3-10. Thunk Models . . . . . 3-13
3-11. Mapping the flat Address Space Using GDT Selectors . . . . . 3-15
3-12. 16-Bit Address Space Mapped to the Flat Address Space . . . . . 3-16
3-13. Using DosSelToFlat . . . . . 3-17
3-14. Using DosFlatToSel . . . . . 3-17
3-15. Data Item Spans 64K Tile Boundary . . . . . 3-18
3-16. Using _Far16 _Pascal to Declare a 16-Bit Function . . . . . 3-19
3-17. Using _Seg16 to Declare 16-Bit Pointers . . . . . 3-19
3-18. Using the #pragma seg16 Directive . . . . . 3-20
3-19. Using _Seg16 . . . . . 3-20
3-20. Using _Far16 in a 32-Bit Module . . . . . 3-21
3-21. Passing Different Size Parameters . . . . . 3-21
3-22. Structure Alignment . . . . . 3-22
3-23. Example of a Packing Problem . . . . . 3-23
3-24. Another Packing Problem . . . . . 3-24
3-25. Encountering Alignment Problems . . . . . 3-24
3-26. An Array Structure Element . . . . . 3-24
3-27. Aligning an Array of Structures . . . . . 3-25
3-28. Using the _Far16 _Pascal and _Seg16 Keywords . . . . . 3-25
3-29. Another Example of Using _Far16 _Pascal and _Seg16 . . . . . 3-27
3-30. Using the Packing Pragma Convention . . . . . 3-28
3-31. Declaring a 16-Bit Function in 32-Bit Code . . . . . 3-29
3-32. Creating a 16-Bit Window From Within a 32-Bit Module . . . . . 3-30
3-33. Passing a 16:16 Pointer as a Message Parameter . . . . . 3-31
3-34. Mixed Model Programming—WinSetWindowThunkProc() Function . . 3-31
3-35. Mixed Model Programming—Thunk Procedure . . . . . 3-32
3-36. Calling 32-Bit Code from 16-Bit Code . . . . . 3-33
4-1. Allocating Memory . . . . . 4-3
4-2. Freeing Memory . . . . . 4-4
4-3. Suballocating Memory . . . . . 4-6

4-4. Using Named Shared Memory . . . . . 4-7
4-5. Giving Unnamed Shared Memory . . . . . 4-8
4-6. Getting Unnamed Shared Memory . . . . . 4-9
4-7. Generating Dynamic Code . . . . . 4-10
4-8. Getting Thread and Process Information . . . . . 4-16
4-9. Using Event Semaphores—16-Bit Version . . . . . 4-20
4-10. Using Event Semaphores—32-Bit Version . . . . . 4-21
4-11. Using Semaphores for Mutual Exclusion—16-Bit Version . . . . . 4-22
4-12. Using Semaphores for Mutual Exclusion—32-Bit Version . . . . . 4-24
4-13. Using Semaphores for Multiple Waiting . . . . . 4-25
4-14. Using Resources . . . . . 4-30
4-15. 32-Bit Code for Finding Directories . . . . . 4-32
4-16. Open Dialog . . . . . 4-42
4-17. SaveAs Dialog . . . . . 4-43
4-18. Font Dialog . . . . . 4-43
4-19. Notebook Control Window . . . . . 4-44
4-20. Control Window Classes, Styles, and Messages . . . . . 4-44
4-21. Container Control Window . . . . . 4-46
4-22. Container Control Window Classes, Styles, and Messages . . . . . 4-46
4-23. Value-Set Control . . . . . 4-48
4-24. Value-Set Control Window Classes, Styles, and Messages . . . . . 4-48
4-25. Slider Control . . . . . 4-49
4-26. Slider Control Window Classes, Styles, and Messages . . . . . 4-49
5-1. Static Linking . . . . . 5-1
5-2. Dynamic Linking . . . . . 5-4
5-3. Resolving Dynamic Link References . . . . . 5-5
5-4. DLL Data . . . . . 5-8
5-5. Specifying when to Execute the Initialization and Termination Function 5-9
5-6. A Simple DLL . . . . . 5-11
5-7. Module-Definition File for Simple DLL . . . . . 5-11
5-8. Compiling and Linking a Simple DLL . . . . . 5-11
5-9. Using a Simple DLL . . . . . 5-12
5-10. DEF File for Application Using a DLL . . . . . 5-12
5-11. Compiling and Linking an Application . . . . . 5-12
5-12. Creating an Import Library . . . . . 5-12
5-13. Linking with an Import Library . . . . . 5-13
5-14. Specifying Data Segments with Different Attributes . . . . . 5-13
5-15. Prototype for the _DLL_InitTerm Function . . . . . 5-14
5-16. Prototype for the _CRT_init Function . . . . . 5-14
5-17. Prototype for the _CRT_term Function . . . . . 5-14
5-18. Specifying the System Linkage . . . . . 5-14
5-19. A DLL Initialization Function Entry Point . . . . . 5-15
5-20. A DLL Initialization Function . . . . . 5-15
5-21. Using A Runtime Dynamic-Linked Library . . . . . 5-19
5-22. A 32-Bit DLL . . . . . 5-20
5-23. Enabling Memory Protection for DLLs . . . . . 5-20
5-24. Accessing Protected DLL Data . . . . . 5-21
5-25. Using the PROTECT16 Parameter . . . . . 5-22
6-1. Enhanced DOS Session System Structure and Control Flow . . . . . 6-3
6-2. Loading VEMM . . . . . 6-7
6-3. Memory Map of Areas Supported by Extended Memory . . . . . 6-8
6-4. Loading VXMS . . . . . 6-9
6-5. Loading DPMI . . . . . 6-12
6-6. Structure of Virtual and Physical Device Drivers . . . . . 6-14
6-7. Physical and Virtual Device Drivers Under OS/2 2.0 . . . . . 6-16
7-1. Generic Stack Functions in C . . . . . 7-2

7-2. Using Generic Stack Functions in C . . . . . 7-2
7-3. Polymorphism by Inheritance . . . . . 7-4
7-4. Classes and Metaclasses . . . . . 7-5
7-5. SOM Objects at Initialization . . . . . 7-5
7-6. Class Definition File Template . . . . . 7-7
7-7. Structure of a Sample Class Definition File . . . . . 7-9
7-8. Maintaining Compatibility by Modifying the Release Order . . . . . 7-10
7-9. Syntax of OIDL Comments . . . . . 7-11
7-10. Processing a Class Definition File . . . . . 7-11
7-11. (SOM) Compiling .SC Files . . . . . 7-12
7-12. Class Definition File . . . . . 7-13
7-13. C-Language Source-Program Template for Implementation of Dog Class . . . . . 7-14
7-14. General Form for a Method Stub . . . . . 7-14
7-15. Prototype of Method with Parameters . . . . . 7-14
7-16. Stub of Method with Parameters . . . . . 7-14
7-17. Suppressing SOM Tracing . . . . . 7-15
7-18. C-language Source Program for Implementation of Dog Class . . . . . 7-15
7-19. Underscored-Data-Name macro . . . . . 7-16
7-20. Underscored-Method-Name macro . . . . . 7-17
7-21. Invoking Methods with Identical Names but Nonrelated Classes . . . 7-17
7-22. Replacing SOM Functions . . . . . 7-20
7-23. Invoking Methods and Accessing Data in a New Dog Class (DOG.CSC) 7-21
7-24. Implementation of a New Dog Class (DOG.C) . . . . . 7-22
7-25. Client of the New Dog Class . . . . . 7-23
7-26. Defining an Object . . . . . 7-23
7-27. Output from Client of Dog Class . . . . . 7-24
7-28. Inheritance Relationships Between Classes and Subclasses . . . . . 7-24
7-29. LDOG.CSC—LittleDog Class Definition File . . . . . 7-25
7-30. BDOG.CSC—BigDog Class Definition File . . . . . 7-25
7-31. LDOG.C—LittleDog Class Implementation . . . . . 7-25
7-32. BDOG.C—BigDog Class Implementation . . . . . 7-26
7-33. Client of Dog, BigDog, and LittleDog Classes . . . . . 7-27
7-34. Output from Client of Dog, BigDog, and LittleDog Classes . . . . . 7-27
7-35. DOGMETA.CSC—Class Definition for Metaclass of Dog Class . . . . . 7-28
7-36. DOGMETA.C—Implementation of DogMeta Class . . . . . 7-28
7-37. DOG.CSC—Associating a Metaclass with a Class . . . . . 7-29
7-38. Client of Dog and DogMeta Classes . . . . . 7-29
7-39. DOG.CSC—Implied Metaclass for the Dog Class . . . . . 7-30
7-40. Implementation of an Implied Metaclass for the Dog Class . . . . . 7-31
7-41. A Client of the Dog Class with an Implied Metaclass . . . . . 7-32
7-42. Class Relationships in ANIMALS Sample Program . . . . . 7-33
7-43. ANIMAL.CSC—ANIMALS with Implied Metaclasses . . . . . 7-34
7-44. ANIMAL.C—ANIMALS with Implied Metaclasses . . . . . 7-35
7-45. DOG.CSC—ANIMALS with Implied Metaclasses . . . . . 7-38
7-46. DOG.C—ANIMALS with Implied Metaclasses . . . . . 7-39
7-47. BDOG.CSC—ANIMALS with Implied Metaclasses . . . . . 7-42
7-48. BDOG.C— ANIMALS with Implied Metaclasses . . . . . 7-43
7-49. LDOG.CSC—ANIMALS with Implied Metaclasses . . . . . 7-43
7-50. LDOG.C—ANIMALS with Implied Metaclasses . . . . . 7-43
7-51. MAIN.C—Client of ANIMALS with Implied Metaclasses . . . . . 7-44
8-1. Objects in a Folder . . . . . 8-7
8-2. Workplace Object Class Hierarchy . . . . . 8-8
8-3. Adding Pages to an Object's Settings Notebook . . . . . 8-10
8-4. Adding Pages to an Object's Settings Notebook . . . . . 8-11
8-5. Removing a Page from an Object's Settings Notebook . . . . . 8-12

8-6. Pop-Up and Conditional Cascade Menus . . . . . 8-12
8-7. Removing Standard Items from an Object's Pop-Up Menu . . . . . 8-14
8-8. Flags for Standard Pop-Up Menu Items Defined by Other Workplace Classes . . . . . 8-14
8-9. Resource File Defining New Items for Object's Pop-Up Menu . . . . . 8-15
8-10. IDs for Standard Items in a Pop-Up Menu . . . . . 8-15
8-11. Adding Class-Specific Items to an Object's Pop-Up Menu . . . . . 8-16
8-12. Adding an Item to a Pop-Up Menu Submenu . . . . . 8-16
8-13. Removing Class-Specific Items from a Pop-Up Menu . . . . . 8-17
8-14. Conditional Cascaded Menus . . . . . 8-17
8-15. Resource File Defining Pulldown for Object's Pop-Up Menu . . . . . 8-18
8-16. Creating a Conditional Cascaded Menu for a Pop-Up Menu Item . . . 8-18
8-17. Supporting User Selection of New Pop-Up Menu Items . . . . . 8-19
8-18. Details View for a Workplace Class . . . . . 8-23
8-19. A Details Record for an Object . . . . . 8-24
8-20. Format for Items in a Details Record for an Object . . . . . 8-24
8-21. Defining a Structure for CAR Details . . . . . 8-24
8-22. Initializing CLASSFIELDINFO Structures for CAR Details . . . . . 8-25
8-23. Defining Format of Details Data . . . . . 8-27
8-24. Appending Details Data to end of Object's Details Record . . . . . 8-28
8-25. Data Structures for Object In-Use Items . . . . . 8-31
8-26. Example of Object Setup String . . . . . 8-34
8-27. Processing KEYNAMES for a Class . . . . . 8-35
8-28. Changing Existing Objects on the Desktop . . . . . 8-41
8-29. REXX Batch File to List all Classes Registered With Workplace . . . . 8-42
8-30. Installation Program for Workplace Object . . . . . 8-43
8-31. REXX Batch File for Installing Workplace Objects . . . . . 8-45
8-32. .ASSOCTABLE Extended Attributes . . . . . 8-46

# Tables


1-1. OS/2 Function Groups . . . . . 1-29
3-1. OS/2 2.0 Application Types . . . . . 3-2
3-2. OS/2 Version 2.0 Programming Models . . . . . 3-5
3-3. WORD/DWORD Alignment . . . . . 3-23
4-1. 16-Bit to 32-Bit Memory-Management Functions . . . . . 4-2
4-2. 16-Bit and 32-Bit DosSubSetMem . . . . . 4-5
4-3. 16-Bit to 32-Bit Tasking Functions . . . . . 4-12
4-4. 16-Bit to 32-Bit Semaphore Functions . . . . . 4-19
4-5. 16-Bit to 32-Bit Named-Pipe Functions . . . . . 4-27
4-6. 16-Bit to 32-Bit Queue Functions . . . . . 4-28
4-7. 16-Bit to 32-Bit Timer Functions . . . . . 4-28
4-8. 16-Bit to 32-Bit Dynamic-Linking Functions . . . . . 4-29
4-9. 16-Bit to 32-Bit Device I/O Functions . . . . . 4-30
4-10. 16-Bit to 32-Bit File-System Functions . . . . . 4-31
4-11. 16-Bit to 32-Bit Message-Retrieval Functions . . . . . 4-34
4-12. 16-Bit to 32-Bit Code-Page Management Functions . . . . . 4-34
4-13. 16-Bit to 32-Bit Session-Management Functions . . . . . 4-35
4-14. 16-bit to 32-bit Error Management Functions . . . . . 4-35
4-15. 16-Bit to 32-Bit Exception-Management Functions . . . . . 4-36
4-16. 32-Bit VDD Services . . . . . 4-37
4-17. Version 2.0 Print Functions . . . . . 4-39
4-18. OS/2 2.0 Workplace Functions . . . . . 4-40
4-19. New IPF Tags . . . . . 4-41
4-20. Version 2.0 DDF Functions . . . . . 4-41
4-21. Version 2.0 Migration Functions . . . . . 4-42
4-22. Font and File Dialog Functions . . . . . 4-43
4-23. Path, Region, and Bit Map Functions . . . . . 4-51
4-24. Font and Character Functions . . . . . 4-52
4-25. Polyline Functions . . . . . 4-52
4-26. Transformation Functions . . . . . 4-52
4-27. PM Helper Macros . . . . . 4-53
5-1. Why Static Linking? . . . . . 5-2
6-1. OS/2 2.0 Virtual Device Drivers . . . . . 6-15
6-2. VDD Services . . . . . 6-17
7-1. SOM-Supplied Classes . . . . . 7-5
7-2. SOM C-Language Bindings Files . . . . . 7-12
7-3. SOM Environment Variables . . . . . 7-13
7-4. SOM Naming Conventions . . . . . 7-16
7-5. Class-Specific Macros . . . . . 7-17
7-6. Macros and Functions for Manipulating SOM IDs . . . . . 7-18
7-7. SOM Debug Macros and Control Variables . . . . . 7-18
7-8. SOM Error Severity Levels . . . . . 7-19
7-9. Replaceable SOM Functions . . . . . 7-19
7-10. Summary of SOM and Object-Oriented Programming Terminology . . 7-46
8-1. Some Workplace Objects Provided by OS/2 2.0 . . . . . 8-7
8-2. Workplace Storage Classes . . . . . 8-9
8-3. Defining Methods for the WPObject Class . . . . . 8-9
8-4. Settings Notebook Methods . . . . . 8-11
8-5. Pop-Up Menu Methods to Modify an Object's Pop-up Menu . . . . . 8-13
8-6. Flags for Standard Pop-Up Menu Items Inherited from WPObject . . . 8-13
8-7. Pop-Up Menu Methods that Support New Pop-Up Menu Items . . . . . 8-19
8-8. Pop-Up Menu Methods Supporting Standard Pop-Up Menu Items . . . 8-20

8-9. Predefined Open Views for Workplace Objects . . . . . 8-20
8-10. Object Information Methods . . . . . 8-22
8-11. Object Class Styles . . . . . 8-22
8-12. DM Notifications from the Shell to Target Objects and to Windows . . 8-29
8-13. Save/Restore the Object's State Methods . . . . . 8-29
8-14. Object Usage Methods . . . . . 8-30
8-15. Types of In-Use Items for Objects . . . . . 8-30
8-16. Setup/Cleanup Methods . . . . . 8-32
8-17. WPObject KEYNAMES and Values . . . . . 8-32
8-18. Some WPObject Class Methods . . . . . 8-36
8-19. Default Class Styles for Objects . . . . . 8-36
8-20. New Methods for CAR Class . . . . . 8-37
8-21. Workplace API . . . . . 8-40
8-22. Object IDs for Predefined System Folders . . . . . 8-41
8-23. REXX Utility Workplace Functions . . . . . 8-42
8-24. Summary of Workplace Terms . . . . . 8-48

# Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights or other legally protectible rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

## Trademarks and Service Marks

The following terms, denoted by an asterisk (*), used in this publication, are trademarks or service marks of the IBM Corporation in the United States or other countries:

C/2
C Set/2
Common User Access
CUA
IBM
Operating System/2
OS/2
Personal System/2
Presentation Manager
PS/2
SAA
Systems Application Architecture

The following terms, denoted by a double-asterisk (**) used in this publication, are trademarks of other companies as follows:

| | |
|---|---|
| Intel | Intel Corporation |
| Windows | Microsoft Corporation |
| Microsoft | Microsoft Corporation |
| Lotus | Lotus Development Corporation |
| 123 | Lotus Development Corporation |
| Windows/286 | Microsoft Corporation |
| Helvetica | Linotype Company |
| Courier | Linotype Company |

## Double-Byte Character Set (DBCS)

Throughout this publication, you will see reference to specific values for character strings. The values are for single-byte character set (SBCS). If you use the double-byte character set (DBCS), note that one DBCS character equals two SBCS characters.

# About This Book

This book is a companion to the *OS/2 Programming Guide,* Volumes I—III. It describes specific aspects of programming in the OS/2 2.0 environment. The topics in this book reflect the flexibility of OS/2 2.0.

## Who Should Read This Book

This book is intended for the professional programmer with some knowledge of the C programming language. It is intended for the new OS/2 programmer migrating from DOS or Windows, as well as for the experienced OS/2 programmer migrating from the 16-bit environment.

## How This Book is Organized

Chapter 1, "OS/2 2.0 Overview" describes the highlights of OS/2 2.0, the benefits of OS/2 2.0 and Presentation Manager, the fundamental concepts of the base operating system, or Control Program, and the fundamental concepts of Presentation Manager.

Chapter 2, "The 32-bit OS/2 Programming Environment" describes the 80386 architecture and the OS/2 utilization of the 80386.

Chapter 3, "The Application Development Environment" describes the type of applications that can run under OS/2 2.0, the programming models for OS/2 2.0 applications, the OS/2 2.0 program development environment, and migration issues.

Chapter 4, "Comparison of 16-Bit and 32-Bit OS/2 Functions" describes the differences between the 16-bit and 32-bit OS/2 functions for the Control Program and Presentation Manager.

Chapter 5, "Dynamic Linking" describes static and dynamic linking, DLL data, DLL initialization and termination, building a DLL, and protected DLLs.

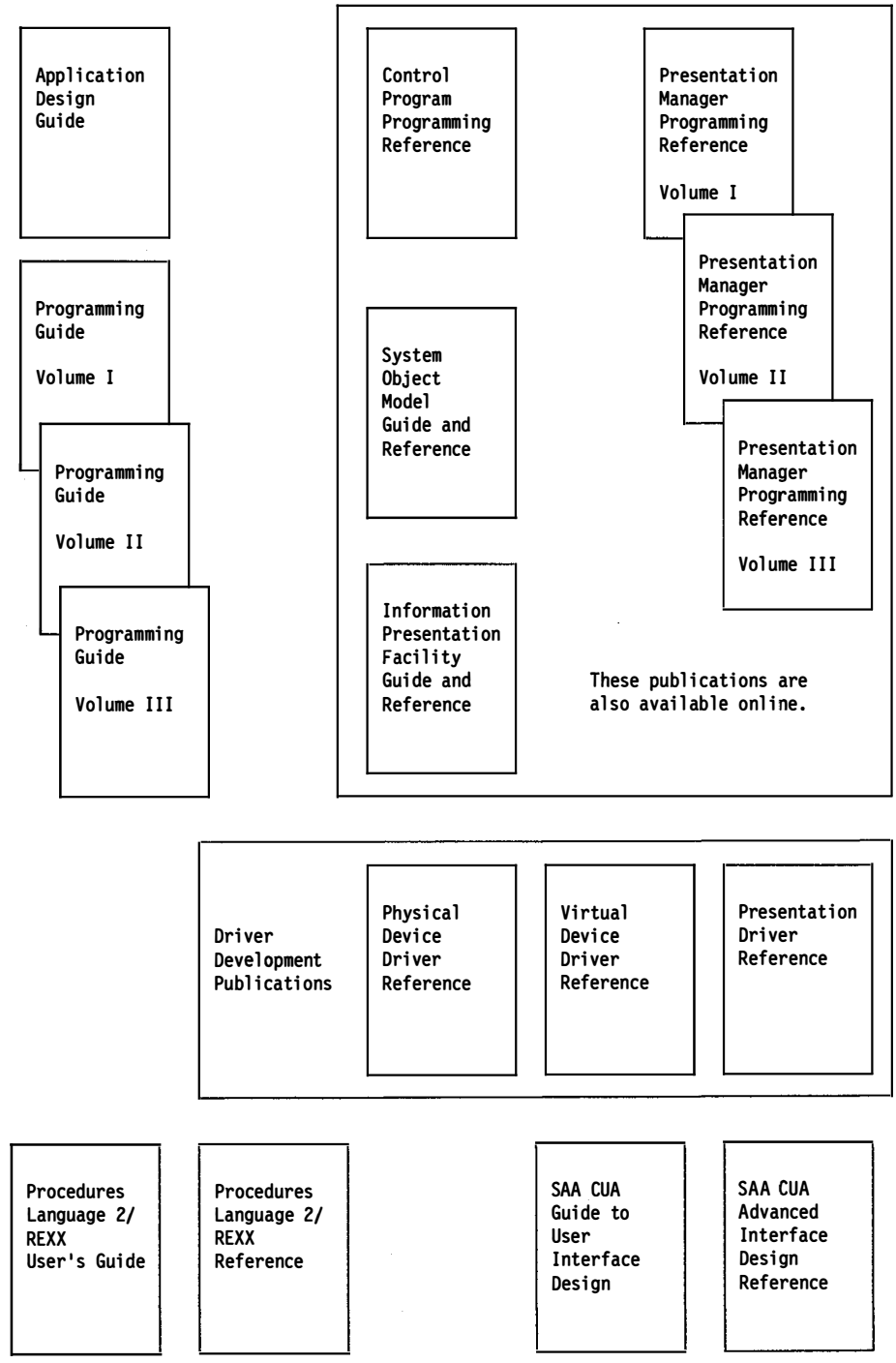
Chapter 6, "Multiple Virtual DOS Sessions" describes the differences between the real mode OS/2 DOS Box and Enhanced DOS Session, how I/O devices are virtualized by OS/2 virtual device drivers, how Enhanced DOS Session takes advantage of the flat memory model, and how Enhanced DOS Session provides compatibility for DOS applications.

Chapter 7, "Object-Oriented Programming Using SOM" describes the IBM System Object Model environment.

Chapter 8, "Workplace Programming Interface" describes the OS/2 2.0 Workplace Programming Interface.

Appendix A, "Sample Programs Cross Index" shows the OS/2 API functions, and the sample programs that demostrate their use.

## OS/2 2.0 Technical Library



# Chapter 1. OS/2 2.0 Overview

This chapter describes:

- Highlights of OS/2* 2.0*
- Benefits of OS/2 2.0 and the Presentation Manager* (PM) program
- Concepts of the base operating system
- Concepts of PM
- Concepts of the Graphics Programming Interface

## OS/2 2.0 Highlights

OS/2 2.0 is an advanced multitasking, single-user operating system for personal computers. Like previous versions of the operating system, OS/2 2.0 provides an application programming interface (API) that supports multitasking, multiple threads, dynamic linking, interprocess communication, a graphical user interface, and a graphics programming interface. Features available in previous versions of the operating system, such as a high-performance file system, extended file attributes, and long file names, also are available in OS/2 2.0.

New features of OS/2 2.0 include:

- Full use of the the Intel** 80386 processor features, such as flat memory modeling, paged virtual memory, and enhanced instruction set
- Portability between different processing platforms
- Full compatibility with Version 1.X
- Execution of multiple DOS sessions
- Virtual device drivers for supporting virtual devices to the DOS environment

### 386 Features

Unlike previous versions of the operating system, OS/2 2.0 takes advantage of features of the Intel 80386 processor, such as the flat memory model, paged virtual memory, and enhanced instruction set. This means that OS/2 applications do not manipulate selectors and offsets as they do in the segment memory model of previous versions of the operating system. Instead, OS/2 2.0 applications view memory as a large, linear, address space addressable by 32-bit offsets from the beginning of memory. Paged virtual memory means that memory is managed more efficiently, and the enhanced instruction set lets the application handle 32-bit values in single instructions.

### Portability

A benefit of the flat memory architecture of OS/2 2.0 is application and operating-system portability. Programs written for other operating systems that use a 32-bit, linear (flat) memory model are easily portable to OS/2 2.0. OS/2 applications that use the linear memory model will also be easily portable to future versions of OS/2 2.0 (and other operating systems) that run under other microprocessors. The flat model architecture is easily portable to most processing

platforms, because hardware must provide only a base register capable of addressing a large-paged linear address space and an offset register for indexing into the address space. OS/2 2.0 was not designed to be 386-specific, but rather a 32-bit OS/2 operating system implemented on the 80386 platform.

## Compatibility with Version 1.X

OS/2 2.0 requires the features of the 80386; it does not run on computers that use the Intel 80286 processor. This means that applications developed for OS/2 2.0 cannot run under Version 1.X of the operating system. However, applications developed for Version 1.X can run under OS/2 2.0, although they won't be able to take advantage of the full features and performance of the operating system.

Although OS/2 2.0 provides a complete set of 32-bit functions, the operating system supports calls to 16-bit functions. This means that some 32-bit functions are converted by the system to 16-bit functions. The code that does the converting is called the *thunk layer*. The thunk layer converts 32-bit parameters to 16-bits, and maps linear addresses to segmented addresses. The functions of the thunk layer are transparent to application developers, and should be of no concern unless you want to develop your own thunk layer for 16-bit dynamic link libraries (DLLs) that are called by 32-bit and 16-bit applications.

## Multiple DOS Sessions

The 80386 processor makes it possible for OS/2 2.0 to manage more than one DOS application at a time. In the virtual 8086 mode of processor, each DOS application runs in its own copy of a DOS environment. This feature of OS/2 2.0 is referred to as Enhanced DOS Session.

## Virtual Device Drivers

The DOS environment in OS/2 2.0 provides an extendable architecture that allows the DOS environment to be tailored. At the heart of this extendibility is the virtual device driver architecture. All of the low-level DOS support, which in previous versions of the OS/2 2.0 resided in physical device drivers, has been moved into virtual device drivers. Because of the virtual 8086 mode, all interrupt processing is done in protect mode. Bimodal device drivers are no longer needed. The new driver architecture provides physical device drivers for basic device support and virtual device drivers for supporting virtual devices to the DOS environments. Conceptually, a virtual device driver provides the mechanism for sharing a physical device between OS/2 protect-mode applications and DOS applications.

# The OS/2 Operating System and Presentation Manager Program

In a multitasking environment, it is important that all applications have some portion of the screen through which to interact with the user. One of the principal goals of OS/2 2.0 is to provide visual access to most, if not all applications at the same time. This access can be granted by giving selected applications full use of the screen while others wait in the background, or by letting applications share the screen. In OS/2 2.0, the *session* in which the application runs dictates whether the application receives complete control of the screen or must share it with other applications.

Depending on the application type, the system can start an application in a full-screen session. An application running in a full-screen session has complete control of the screen. Applications that have complete control of the screen are called *full-screen applications*.

When the operating system first starts, it creates the PM session. PM manages this session, as well as the screen, keyboard, and mouse resources used by applications running in this session. Applications running in a PM session are called *PM applications*. PM applications share the screen, keyboard, and mouse resources with other PM applications.

Each PM application is allocated a portion of the screen, called a *window*. PM applications allow multiple windows to be displayed on the screen at a time. In this way, PM applications share the screen.

A window is the user interface for an application. It can contain menus, scroll bars, and other controls through which the user communicates with the application.

A PM application must create its own window before producing any output or receiving any input. The operating system provides the application with detailed information about what the user is doing with the window and automatically carries out many of the tasks the user requests, such as moving and sizing the window.

A PM application can create and use any number of windows to display information in a variety of ways. PM manages the screen, controlling the placement and display of windows, and ensuring that no two applications attempt to access the same part of the screen at the same time. (In such a case, PM overlaps the window of one application with the window of the other.)

## Queued Input

In traditional programming environments, an application reads from the keyboard by making an explicit call to a function (GETCHAR, for example). The function typically waits until the user presses a key before returning the character code to the application. A PM application does not make explicit calls to read from the keyboard. Instead, the operating system receives all input from the keyboard, mouse, and timer into the system queue. Each input is then redirected to the application by copying it from the system queue to the application queue. When the application reads the input, the next input is redirected to the application queue, and so on. As the application reads each input, it dispatches messages to the appropriate windows.

In a PM application, input from the keyboard and mouse is provided automatically to every window that is created. The operating system provides input in a uniform format called an input message. The message contains information about the input that far exceeds the information available in other environments. It specifies the system time, the position of the mouse, the state of the keyboard, the scan code of the key (if a key is pressed), the number of the mouse button (if a button is pressed), and the device that generated the message. For example, the keyboard message WM_CHAR corresponds to the pressing or releasing of a specific key. In each message, the operating system provides a device-independent virtual-key code that identifies the key, in addition to the device-dependent scan code generated by the keyboard. The message also specifies the status of other keys on the keyboard, such as SHIFT, CTRL, and NUMLOCK. Keyboard, mouse, and timer messages all have the same format and are processed in the same manner.

## Device-Independent Graphics

In PM operations, you have access to a set of device-independent graphics operations. This means that your applications can easily draw simple or complex shapes; they can make the same calls and use the same data to draw a high-resolution graphics display that they would use to draw on a dot-matrix printer.

The operating system requires presentation drivers to convert graphics-output requests to output for a printer, plotter, display, or other output device. A presentation driver is an executable library that is loaded by PM and is used to carry out graphics operations in the *context* of the specific device; that is, the device driver, the output device, and the communications port.

## Shared Resources

OS/2 2.0, a multitasking system, requires that PM applications must share the display, the keyboard, the mouse, and even the processor, with all other applications that are currently running in the same session, or in other sessions. For this reason, the operating system carefully controls these resources and requires applications to use a specific programming interface that guarantees this control.

# Control Program Fundamentals

The control program consists of all the OS/2 functions that are used to create processes and threads, access disk files and devices, allocate memory, and retrieve or set information about the system. In PM applications, these functions typically are used to carry out tasks for which no corresponding window-manager or graphics programming interface function exists. Full-screen applications almost exclusively use these functions, even to interact with the user and access the devices of the computer.

## Multitasking

*Multitasking*, one of the principal features of OS/2 2.0, is the ability of the system to manage the execution of more than one application at a time. This ability helps to optimize use of the computer, because time normally spent by an application waiting for user input is distributed to other applications that might be printing a document or recalculating a spreadsheet.

OS/2 2.0 supports two types of multitasking. The first type enables an application to start other programs, in separate processes, that will execute concurrently with the application. These programs can be a new copy of the application, a related program that is designed to work with the application, or an unrelated program. The second type of multitasking enables applications to run multiple threads of execution within the same process; separate activities can be multitasked within the application.

Although all the applications appear to the user to be running simultaneously, the processor actually can perform only one task at a time. The processor often must wait for input or output from a relatively slow device, such as the keyboard or disk. While the processor is waiting, another task can be run.

An application often has to perform multiple tasks that can be synchronized. To make efficient use of the processor, an application can be structured to allow one piece of work to be executed while another is waiting for something to happen. The application can be written so that it can be executed with other applications of which it has no knowledge.

The operating system manages the concurrent processing of applications by making certain that computer resources are so allocated that one application does not interfere with another. However, to share the OS/2 environment, the application must plan and synchronize execution of its routines.

To co-exist harmoniously in the multitasking environment, each application must be able to communicate with other applications, to synchronize its activities, and to serialize its use of resources. It must be able to start, stop, and control its units of execution.

The operating system helps to control the multitasking environment on three levels: sessions, processes, and threads. This three-tier hierarchy addresses both the user's need to control concurrent execution, and the developer's need to design applications whose activities can be performed concurrently.

Figure 1-1 illustrates the hierarchical relationships of the multitasking elements.

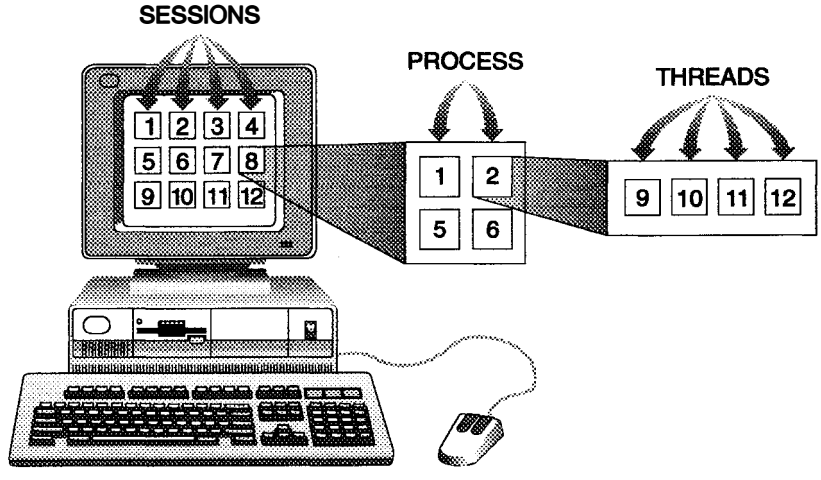



*Figure 1-1. Multitasking Hierarchy*

## Sessions

A *session* is the highest level of the OS/2 multitasking hierarchy. It consists of virtual/logical devices, such as the screen (or a window), keyboard, and mouse, and their related processes. Logical devices are mapped to these physical devices when the user selects a session by bringing it to the foreground to perform I/O. Generally, each application runs in its own session. However, more than one application in the PM session can share the physical screen, keyboard and mouse.

Sessions can be arranged in a parent-child hierarchy. By default, a parent session can make a child session available for the user to select from the Workplace Shell. A session can start another independent session. In this case, the starting session cannot control whether the user can select the second session. If a session is running in the foreground, it can bring any of its child sessions to the foreground. A parent session can end its own child sessions, but cannot end their descendants. However, if a child session is ended, the operating system automatically ends the related descendant sessions.

## Processes

A *process* is the logical unit of resources allocated to run an application. These resources include memory, files, pipes, queues, system semaphores, and device monitors. Each application that has been loaded into memory and is running is called a *process*. Each session contains at least one process.

One process can create other processes, which are arranged in parent-child relationships similar to sessions. A *child process* created by a *parent process* may inherit access to the handles to files, pipes, and devices owned by the parent process. The parent process retains control of its child processes and any other processes called in the line of descendancy. A process can control the execution of its child processes, and can end itself and its child processes.

A process can give control to a set of routines that are to be executed when the process ends. The routines, called *exit lists*, free allocated resources and perform "general housekeeping" after a normal or abnormal end.

## Threads

The smallest unit of operation to be performed within a process is called a *thread*. A thread consists of instructions, related CPU register values, and a stack. A process always has at least one thread, called the *main thread* or *thread 1*, and can create more threads. These additional threads are useful for carrying out tasks unrelated to the processing of the main thread. For example, a process might create a thread to write data to a disk file. This frees the main thread so that it can continue to process user input. A thread does not own system resources but shares the resources owned by the process that created it. An application runs when the operating system gives control to a thread in the process. Threads are not organized hierarchically. Each thread within a process is a *peer* with the other threads within the process.

Using the API functions, a thread can start other threads within a process. A thread also can temporarily suspend and resume execution of other threads. This can enable the thread to access a time-critical resource. Although all threads in a process are considered to be peers, if the first thread in a process ends, all other threads it created also end. It is necessary to have a thread issue an API function to end itself when it completes its execution.

***Dispatching Priority:*** The operating system assigns each thread a *dispatching priority*, unless one is explicitly defined by the API function. Processor time is allocated to only one thread at a time. The operating system uses *time slicing* to ensure that threads of equal priority are given equal chances for execution. The operating system can preempt a thread when its time slice expires or if a thread with a higher or equal priority is ready for execution. The default minimum time slice is 32 milliseconds. OS/2 timer services, accessed through the API, support all timer-related activities, including synchronizing the activities of threads.

When a thread is created, its priority class, and priority level within the class, are the same as that of the thread that created it. The API gives the application the ability to query or change a thread's priority class and level. The priority classes are *time-critical*, *fixed-high*, *regular*, and *idle-time*. For each of these classes, there are 32 priority levels: 0 to 31. A thread with priority level 31 always receives a time slice before a thread with priority level 30, and so on. Threads are dispatched as follows:

| Priority class | When the thread is dispatched |
|---|---|
| **Time critical** | Immediately and continuously from within the priority level. (These threads handle communications or real-time applications.) |
| **Fixed high** | When no time-critical threads are waiting. |
| **Regular** | Based on display status (background or foreground), recent I/O activity, and processor use. Most threads are classified with regular priority. |
| **Idle-time** | When no regular, fixed-high, or time-critical threads are waiting. |

Although an application can set the priority level of a thread at any time, only applications that use more than one thread or process should do so. The best use of priority is to speed up threads on which other threads depend. For example, an application might temporarily raise the priority of a thread loading a file if another thread is waiting for that file to be loaded. Because the priority of a thread is relative to all other threads in the system, raising the priority of the threads in an application merely to get the extra CPU time adversely affects the overall operation of the system.

## Dynamic Linking

Dynamic linking enables an application to gain access at run time to functions that are not part of its executable code. These functions are contained in dynamic link libraries (DLLs)—modules that contain executable code but cannot be run as applications. Instead, applications load the appropriate DLLs and execute the code in the libraries by linking to them dynamically.

DLLs are very common in OS/2 2.0. In fact, most of the operating system is contained in them. The chief advantage of DLLs is that they reduce the amount of memory needed by an application. An application loads a DLL only if it needs to execute a function in the DLL. Once the DLL is loaded, the system also shares it with any other applications that need it. This means that only one copy of the DLL is loaded at any one time.

## Memory Management

All 32-bit applications can, at any time, allocate additional memory for their own use. In Version 1.X of the operating system, applications request memory by *segments*. A segment is a unit of virtual and physical memory allocation based on the Intel 286 processor architecture. In OS/2 2.0, applications request memory by *object*. An object is the unit of memory allocation in a 32-bit, flat architecture.

An memory object is allocated in units of 4KB. One 4KB unit is called a page. Each page within a memory object can be in one of two states, either uncommitted (that is, the linear address range has been reserved, but is not yet backed by physical storage) or committed (physical storage as been allotted for the logical address range).

In OS/2 2.0, code and data required for relatively immediate processor execution is kept in physical memory. Code and data which is not required for immediate processor execution is kept on external storage devices (swap space).

Functions that allocate memory return 32-bit pointers to memory objects ranging in size from 1 page (page = 4KB) to any size supported by available swap space. All pointer references are 32-bit near pointers. No segment register loads are involved; thus all of the segment registers are equal: CS = DS = SS = ES. Virtual memory works by demand paging, rather than by compaction and segment swapping. This has important implications for sizing memory objects in order to gain optimum system performance. All memory allocations, whether private or shared, are guaranteed to fill committed pages with zeros. Application developers can rely on this fact when determining the initial contents of memory.

OS/2 2.0 protects memory from unauthorized use. The process that allocates memory owns that memory, and no other process can access it. Attempting to access memory owned by another process causes a protection violation and usually ends the process.

If two processes need to share memory, one can create a shared memory object and either pass the pointer to the process that is to share the memory, or pass the name of the shared memory to that process. The sharing processes must manage the shared memory.

## The File System

The OS/2 file system enables an application to organize and maintain data on external devices, and provides a logical view of the storage media. This enables applications to manipulate data without having to be familiar with the characteristics of each device.

The operating system has two file systems: the *file allocation table* (FAT) file system and the *High Performance File System* (HPFS). The FAT file system is the default file system for the operating system, and does not need to be installed. HPFS can be installed during system initialization. HPFS manages large disk media in a fast and consistent manner and supports files with the long file-name format.

The file system is arranged in a hierarchy on the physical disk. This disk can be subdivided into two or more logical disks, or *partitions*, each with its own hierarchy.

The basic element in the file system is the *file* itself. A file represents a serial stream of characters. A hierarchical collection of files is called a *directory*. A directory hierarchy contains at least one *root directory* at the top of the hierarchy on each logical disk. Directories also can contain other directories, called *subdirectories*, which are collections of files at a lower hierarchical level. Using directories and subdirectories, an application can organize the contents of a disk into logical groups of files.

The operating system provides API functions to enable the application to create, open, read, write, and close files, and to create and delete subdirectories. Once a file is opened, it is assigned an identifier called a *file handle*, which can be used by related processes to refer to the file.

OS/2 2.0 also supports extended attributes, which enable an application to attach various information to a file object. Extended attributes consist of an ASCII text name and an arbitrary value. For information about naming extended attributes and defining data types, see the *OS/2 Programming Guide, Volume I*.

Because any application in the multitasking environment can issue API functions, it is possible that two applications can request access to the same file at the same time. To prevent such conflicts, the operating system provides facilities that, upon request, control access to the information stored in files. The process that opens a file can define how other processes must share the file. If the file is large and needs to be used frequently, a procedure called *file locking* permits a process to protect only small portions of the file, leaving the rest of the file available.

OS/2 2.0 manages its disk files and devices in essentially the same way. For example, an application can use the same functions to open and read from a disk file as it uses to open and read from a serial port. Each open file or device is identified by a unique file handle. The application uses the handle in system functions to access the file or device.

A standard device represents a stream of characters, much like a file does. In most cases, devices look like files to the application. Like files, devices are identified with ASCII names. The *OS/2 Programming Guide, Volume I* describes specific naming conventions.

The operating system provides file subsystem API functions that enable the application to create, open, read, write, and close devices. I/O performed on character devices (devices that handle data one character at a time) must be processed serially. When a device is opened, it is assigned an identifier called a *device handle*. The device handle can be used by a process for input and output, and for other read and write operations to and from the device.

When a process opens a file, it specifies whether the file can be shared—that is, whether it can be accessed and possibly modified by other processes. This sharing also applies to devices that a process might open. Processes can open any device directly, including the parallel port and the serial ports. OS/2 2.0 provides a wide range of I/O control functions that a process can use to access and set the modes of the devices it has opened.

Each process started in the OS/2 environment is provided with a standard set of device handles. These identify the system's standard input device (usually the keyboard), the system's standard output device (usually the screen), and the standard error device (usually the screen) to which error information is written.

Ordinarily, the system automatically opens three file handles when an application starts: the standard-input, standard-output, and standard-error files. These files handles correspond to the keyboard and full-screen display. The application can use these file handles to read from the keyboard and write to a full-screen display.

An application cannot distinguish between file handles, device handles, and pipe handles. The advantage of using the standard device handles is that data stored in files can be redirected to devices without intervention by the application. This also means that one application can redirect its output data stream to another application's input data stream by using a pipe.

## Interprocess Communication

In a multitasking environment, processes and threads need to communicate with one another to synchronize events and to control access to shared resources. The operating system provides a set of interprocess communication (IPC) protocols for such purposes. These protocols are semaphores, pipes, queues, shared memory, exception handling, multiple DOS sessions, and device support.

### Semaphores

A semaphore is used by a process to signal the beginning and end of a given operation, and to prevent more than one thread within the process from accessing a specific resource at the same time. A process can create and use three types of semaphores: mutual exclusion (mutex), event, and multiple wait (muxwait).

A mutex semaphore is used by several threads within a process, or by several processes, to protect access to critical regions. For example, a mutex semaphore can be used to prevent more than one thread at a time from updating a file on disk.

An event semaphore provides a means for signaling among threads or among several processes. A typical use would be managing shared memory. For example, process 1 writes into the shared region, then uses an event semaphore to signal processes 2 and 3 so that they can proceed to access the shared data.

A muxwait semaphore enables a thread to wait on several event or mutex semaphores simultaneously. It is a compound semaphore that consists of up to 64 event semaphores or mutex semaphores (the two types cannot be mixed). A typical use would be when a thread requires access to several shared regions of memory at once. The system blocks the thread until the thread acquires ownership of all mutex semaphores protecting the shared regions. The thread can then access the regions.

### Pipes

Pipes enable two processes to communicate. They are identified by *handles* and can be accessed like files. To communicate, processes open a pipe, then one retrieves the pipe's *read handle*, and the other retrieves its *write handle*. The processes then communicate by writing to and reading from the pipe using the handles. Typically, a pipe is used to direct the output from one process to the standard input of another process.

In OS/2 2.0, there are two types of pipes— *named* and *unnamed*. An unnamed pipe is used only for communications between related (parent and child) processes. A named pipe can be used to communicate between unrelated processes running on the same or different computers. Any process that knows the name of the pipe can open and use a named pipe. One process (the server process) creates and connects the pipe, and another process (the client process) opens the pipe. Once the pipe is connected and opened, the server and client processes can pass data back and forth.

The client process can be local (on the same computer as the server process) or remote (client process connects to a server process across a local area network, or *LAN*).

## Queues

A queue is an ordered list of data that a process can use to receive information from other processes. Processes pass information to a queue in the form of elements. The process that owns the queue can then read the elements from the queue. (PM applications also have queues, called *message queues*. Message queues and the queues described here are not the same.)

Elements in a queue can be accessed individually. They can be accessed in first in, first out (FIFO) or last in, first out (LIFO) sequence. The application also can define the priority of a queue element. Although multiple processes can write to a queue, only the process that created the queue can read it. Individual elements in a queue can be examined without being removed from the queue.

## Shared Memory

OS/2 2.0 enables two processes, or all processes, in the system to share a single memory object. Applications must explicitly request access to shared memory; the shared memory is protected from applications that are not granted access.

For two processes to share memory, one process allocates a memory object and designates that it is to be shared. The process then issues a function to notify the system that a second process can share the memory. The operating system returns the *linear address*, which is a a unique value that identifies the memory object. The first process then passes the linear address to the second process by way of some type of IPC, such as a queue.

If all processes in the system need to share a memory object, the allocating process gives the object a name. This name then is passed to other processes using IPC, such as a queue or a pipe. The other processes then request access to the object by name, and the operating system returns the selector to the memory object. This technique is called *named shared memory*.

## Exception Handling

With OS/2 2.0, an application can deal internally with unexpected errors (such as memory protection violations and termination signals from other processes) without having to terminate.

A multitasking operating system such as OS/2 2.0 must manage applications carefully. A serious error (such as an attempt to access protected memory) occurring in one application cannot be allowed to damage any other application in the system. To deal with errors that might damage other applications, OS/2 2.0 defines a class of error conditions called *exceptions*, and defines default actions for those errors.

When an exception occurs, the operating system usually ends the application causing the exception, unless the application has registered its own exception-handling routine. An application can use these routines to attempt a recovery from unexpected events. The application might be able to correct the exception and continue.

## Multiple DOS Sessions

OS/2 2.0 provides DOS compatibility through the virtual 8086 mode of the 80386 microprocessor, and it provides more than one DOS compatibility environment. The DOS environment offered with OS/2 2.0 is more DOS-compatible than the one offered with previous versions, because it encapsulates the entire DOS environment in a virtual machine. OS/2 also enables the execution of multiple concurrent DOS sessions. This method of implementation provides pre-emptive multitasking for

DOS sessions, and allows normal OS/2 levels of memory protection; that is, provides isolation of system memory and memory belonging to other applications, protection from illegal memory accesses by unpredictable applications, and the ability to terminate sessions where applications are "hung."

A DOS application can be run in a full-screen, as a window, or as an icon in the background. In addition to better protection, better compatibility, and more DOS sessions, the DOS environment of OS/2 2.0 gives applications more than 600KB in which to execute.

## Device Support

Device drivers are the software interface between applications and the operating system, and the hardware. Device drivers enable the applications and operating system to be somewhat device independent. If the device-driver layer did not exist, the operating system would have to be rewritten every time a new device is added.

There are two types of devices—*character* and *block*. A character device handles streams of data, and performs in a serial manner. A block device normally holds large amounts of data that can be accessed either serially or randomly. Block devices include disks and virtual disks. From the application's viewpoint, transfers of blocks of data are transparent. The application performs I/O using the file-system API functions.

In the multitasking environment, multiple applications share resources, including devices. A device driver manages the device to ensure that the applications have access to it. The application uses an API function to communicate with the operating system. The operating system uses an I/O control (IOCtl) interface to communicate with the device driver.

The device driver often needs access to system services, such as memory or interprocess communication mechanisms. These services are provided by device-helper (DevHlp) routines. Some of the services are:

- Managing the device queue
- Synchronizing threads
- Managing memory
- Managing processes
- Managing interrupts
- Handling the timer
- Monitoring data buffers.

The architecture of the device driver model for OS/2 2.0 separates DOS support from the basic OS/2 device support by providing two types of device drivers: physical and virtual. Most 16-bit OS/2 device drivers work "as is," unless they provide support for the DOS environment using the DevHelp function, SetROMVector. In these cases, the driver still works, but not for the DOS environment. The code written as bimodal code in these drivers still works in OS/2 2.0, but only the protect-mode code will be called.

The basic device driver-model for OS/2 2.0 has also been enhanced to support devices in a paged environment. Most devices that use direct memory access (DMA) cannot handle blocks of memory that are not adjoining, which occurs as soon as paging is enabled. Therefore, a device driver should be written according to the device's ability to deal with a paged environment.

## PM Fundamentals

PM provides a message-based, event-driven, graphical user interface for the OS/2 environment. Some of the major features of PM are:

- The window environment
- The user interface
- Input management
- Application resource management
- Data exchange
- The Information Presentation Facility
- Presentation drivers

PM enables programmers to build applications that conform to Systems Application Architecture* (SAA*) guidelines. For more information on SAA requirements, see the *Systems Application Architecture: Common User Access Guide to User Interface Design* and *Systems Application Architecture: Common User Access Advanced Interface Design Reference*.

## The Window Environment

The PM user interface is based on *windows*, an area of the screen through which interaction is presented to the user. A large number of API functions (which begin with the prefix Win) are available for controlling windows. These functions enable an application to create, size, move, and control windows and their contents. The *OS/2 Programming Guide, Volume II* describes common programming techniques for managing the window environment.

### Defining Window Relationships

A window is an area of the screen where an application displays output and receives input from the user. A screen can have more than one window. The common analogy is that multiple windows on the screen are like many pieces of paper on a desktop. In the analogy, the desktop is the area that comprises the background of the screen. Windows, like papers, can be arranged to lie on top of one another and to overlap. If they overlap, the bottom papers can be either partially or completely hidden. Windows can be defined in the window hierarchy using the API.

Figure 1-2 on page 1-14 illustrates the window hierarchy as it appears on the screen.

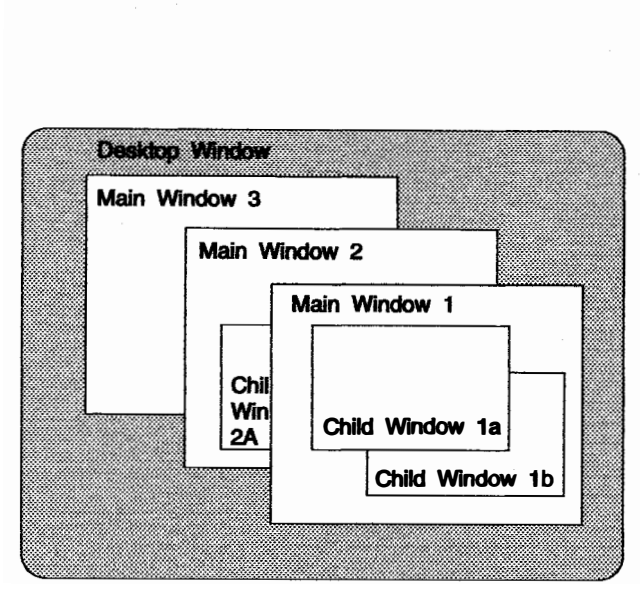



*Figure 1-2. Windows on the Screen*

The *desktop window* is at the top of the hierarchy. Below the desktop window are the top-level windows, called *main windows*. Main windows can overlap one another and, at times, a main window can be completely hidden. Operations on one main window do not affect those on other main windows.

Figure 1-3 illustrates the hierarchical arrangement of windows created by the application.

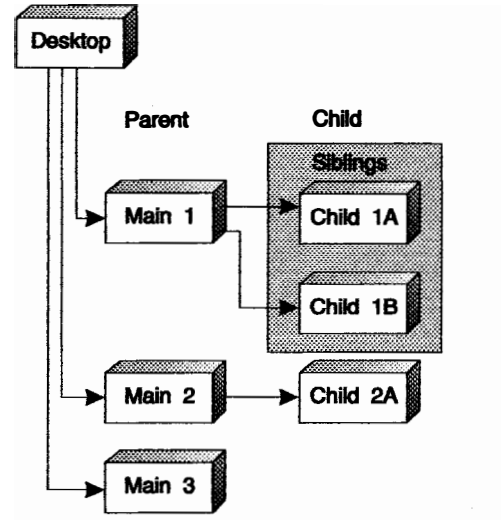



*Figure 1-3. Window Hierarchy*

Main windows can create subordinate windows in a parent-child order of descendancy. A child window is always *clipped* to its parent window, meaning that only the part of a child window that lies within the parent window is visible.

Windows that share the same parent are called *sibling windows*. Like main windows, sibling windows can overlap one another. Every sibling window has a *z order* position that specifies where it lies in the stack of overlapping windows.

The application can define another relationship in addition to the hierarchical one. When a window is created, an *owner window* can be defined. The two windows must be created by the same thread. The owner relationship varies at different levels of the hierarchy. A child window can send messages to its owner window. If one main window owns other main windows, and the owner window is hidden, minimized, or closed, all the owned main windows also are hidden, minimized, or closed.

A window can be visible, hidden, or partly hidden on the screen. When a window is hidden or partly hidden, its size, position, and hierarchical and owner relationships remain the same. However, when the window becomes visible again, any area of the window that was previously hidden is redrawn. A window can also be disabled, meaning that it is still visible but unable to respond to mouse input.

## Creating and Classifying Windows

A window and its associated *window procedure* are considered to be a *program object*. A window procedure "represents" a window in the sense that the window procedure controls all aspects of the window, such as what it looks like, how it responds to changes, and how it processes input. See "Processing Messages" on page 1-19 for more information about the window procedure.

A *window class* is a set of windows that has the same window procedure to implement them. Many windows can belong to a window class. The windows can differ from one another only in the data they process. If multiple applications have need for the same type of window, implementing common window classes is an efficient way of using system resources.

OS/2 2.0 provides many *preregistered* window classes. The windows specified in these classes are designed specifically to meet the needs for a graphics-based standard user interface. These window classes include those that implement such windows as the one described in "Standard and Control Windows" on page 1-16. If a preregistered window class is not provided, the application must register the class at the process level. Several API functions are available for applications to reserve a small area of memory (the *window words* area) for windows in classes registered by the application. If a window is expected to handle large amounts of data, the data should be held in memory and referred from the window words area.

A window class can be defined as *private* or *public*. Windows created in either class can be used by any process in the system. "Public" and "private" refer only to the window class at the time the window is created.

Only the process with which a private class is registered can create a window for that class. The class name must be unique to the process. However, other processes can register private classes with the same class name.

Any process can create a window in a public class. Window procedures for windows in public classes must be available to all processes. Thus, such classes should be defined in DLLs. Public-class names must be unique for each process.

All windows have certain attributes. Each window is identified by a *window handle*. Each window represents a *rectangle* describing the size and position of the window on the screen. The size of a window is defined in picture elements (pels) relative to the origin of the parent window. The origin of a window, the lower-left corner, is the

0,0 coordinate in a set of x and y axes. The x and y coordinates define the top, bottom, and sides of the window. The coordinates range from −32768 to +32767 pels in each direction, so the maximum size that can be specified in any direction is 65535 pels.

The application also can position a window by defining its relative distance from the point of origin (0,0) of its parent window. If the application positions a child window outside its parent window, which is permissible, only the part of the child window within the parent window will be visible on the screen.

Using a set of API functions, the application can modify the behavior of a window in a window class, or create a new class from an existing one. This process, *subclassing*, enables the application to modify the behavior of a single window without rewriting its complete window procedure.

## Providing the User Interface

The Common User Access* (CUA*) is a set of guidelines for designing and writing the application's user interface. The guidelines cover standard menu-bar items, interaction techniques, and window types. Use the CUA guidelines in deciding how to design the user interface for your application's user interface.

Many PM services, if allowed to default, help to enable a consistent user interface among applications designed and written according to the CUA guidelines. Consistency also is enabled by the selection of appropriate options. Ensuring consistency is the responsibility of the application developer.

### Standard and Control Windows

Information is displayed on the screen through the use of windows. PM supports a *standard window*, whose elements generally conform to CUA guidelines.

The standard window, which the application controls using the API functions, can have all or some of the following elements:

- Title-bar icon
- Window borders
- Window sizing buttons
- Menu bar
- Scroll bars
- Window title
- Information area

Figure 1-4 on page 1-17 illustrates a standard window and its elements.

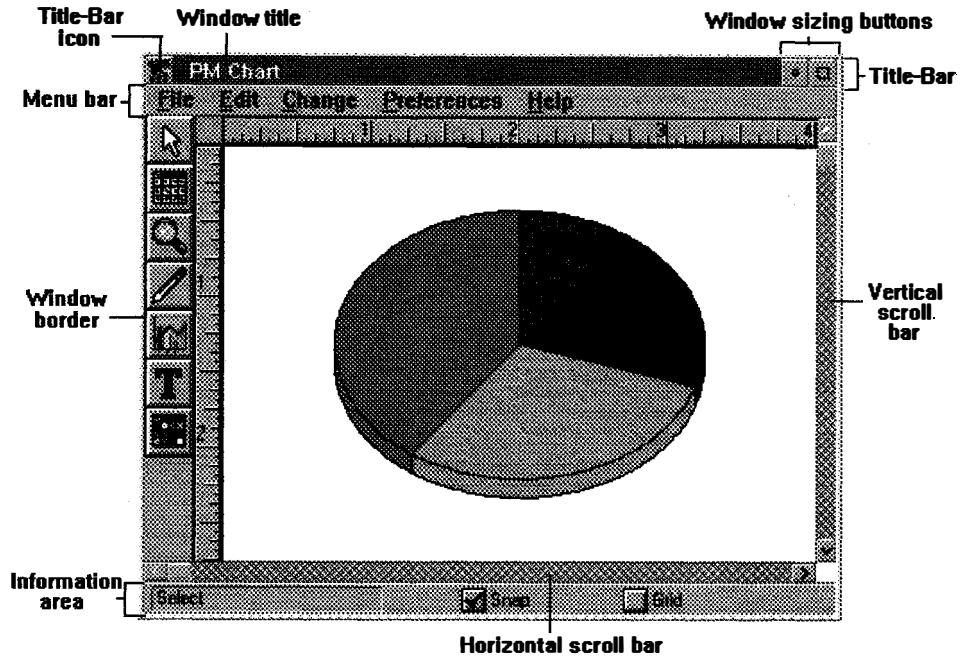



*Figure 1-4. Standard Window*

The title-bar icon, window border, and window sizing buttons enable a user to change the size and position of a window. The menu bar and scroll bars enable a user to work with the window's contents. The window title indicates the name of the object seen in the window, and it also indicates which kind of *view* is displayed. A *view* is a way of looking at an object's information. Different views display information in different forms, which mimics the way information is presented in the real world. The information area displays brief messages to a user about the object or choice that the cursor is on. Information about the normal completion of a process can also appear in the information area. For example, if a user copies several objects from one container to another, the information area in a container's window could display a brief message to tell the user when the copying has been completed.

The standard window is created using a standard *frame window*. The elements of the standard window, such as the title bar and the menu bar, are child windows of the standard frame window. The child windows are called *control windows*. The system maintains a set of preregistered control windows that any application can use to perform I/O.

From the application's perspective, control windows are no different from other windows in the system, and the application can manage them using window-management functions. Each control window has its own window identifier and a specific set of messages. The application can query the system to determine the control window's parent.

Control windows can be used as a part of a *dialog window*. A dialog window can be created using a *dialog template*, which defines the position, appearance, and identifier of the dialog window and each of its child windows. A template can be loaded as a resource or created dynamically in memory. It can be used to create dialog windows of all window classes. The window classes can contain control windows of all window classes. Also, the application can create its own dialog controls by creating and preregistering its own control-window class.

A dialog window is controlled by a window procedure called a *dialog procedure.* The dialog procedure is responsible for responding to all messages sent to the dialog window, either by sending them to the control windows or returning them to the default dialog procedure. A set of API functions enables the application to create, load, process, and cancel dialog windows. The dialog procedure can obtain the handle of its child windows, send messages to them, and process messages and text strings itself.

The standard frame window and the control windows are implemented with standard preregistered window classes. The standard frame window manages the control windows and the client window as the user interacts with them. The frame window also is responsible for routing messages to the appropriate control and client windows.

## Primary and Secondary Windows

CUA guidelines define two types of windows: *primary* and *secondary*. In a PM program, a primary window is a standard window, while a secondary window is a control window or the child of the main window.

A primary window is the main interface point between an object and the user. It appears when a user opens an object, and is used to present a *view* of an object or group of objects when the information displayed about the object or group of objects is not dependent on other objects.

Object information is presented in the area of the window below the menu bar. A user can control the size and position of primary windows on the screen.

A secondary window looks very much like a primary window. For example, both have window borders and title bars. The important distinction between a primary window and a secondary window is based on how they are used. A secondary window is always associated with a primary window and contains information that is dependent on an object in the primary window. A secondary window is used, for example, to allow a user to further clarify action requests. A secondary window is always removed when the primary window is closed or minimized and redisplayed when the primary window is opened or restored.

## Dialog Box

A *dialog box* extends a dialog between a user and a primary or secondary window. It usually appears when the user selects a choice from the menu bar, generating a pull-down menu. Selecting one of the choices in the pull-down menu generates the dialog box. The dialog box can contain buttons, entry fields, icons and text, list boxes, and title bars. A dialog box and its supporting window enable the application to gather input from the user. A temporary dialog window usually is created for special-purpose input, then canceled.

There are two types of dialog boxes, *modal* and *modeless*. A modal dialog box retains control until the application issues a call to cancel it. Users cannot activate other windows belonging to the application until they finish interacting with the modal dialog box. A modeless dialog box enables windows in other applications to be activated after it has been created.

## Handling Mouse and Keyboard Input

The session manager in the operating system manages applications running in the PM environment, including their input and output operations. However, PM handles PM applications, including their input and output operations. PM handles all input as *messages*, which are packets of data.

PM supports user input from the keyboard and *mouse pointer*. The mouse pointer is the symbol associated with the mouse pointing device. Mouse input is provided by pressing a button, and usually is directed at the window under the mouse pointer. The precise position on the screen that is activated is called the *hot spot*. The mouse pointer also can be moved across the screen, and the operating system provides support for that activity. The application can direct all mouse input to a single window called a *mouse capture window*. A mouse capture window enables the application to track all input from the mouse pointer no matter where the mouse pointer is moved on the screen.

Keyboard input is sent when any key on the keyboard is pressed. All keyboard input is directed to one window at a time. The window receiving keyboard input is called the *active window*. A main window, or one of its child windows, is responsible for keeping the window receiving the input visible on the screen.

The *cursor* is a symbol displayed within a window that indicates where characters entered from the keyboard will be placed. The cursor can be moved to any location within a window. Its size and position are defined in coordinates relative to the window in which the cursor is located. The application can create, display, move, and cancel the system cursor.

### Processing Messages

The Common Programming Interface (CPI) defines an application structure that uses a system of queues and application window procedures for processing messages. The PM message system conforms to the CPI and is fundamental to the smooth operation of the PM environment. A complete set of CPI reference manuals is provided in the SAA Library.

Figure 1-5 illustrates the PM message-processing system.

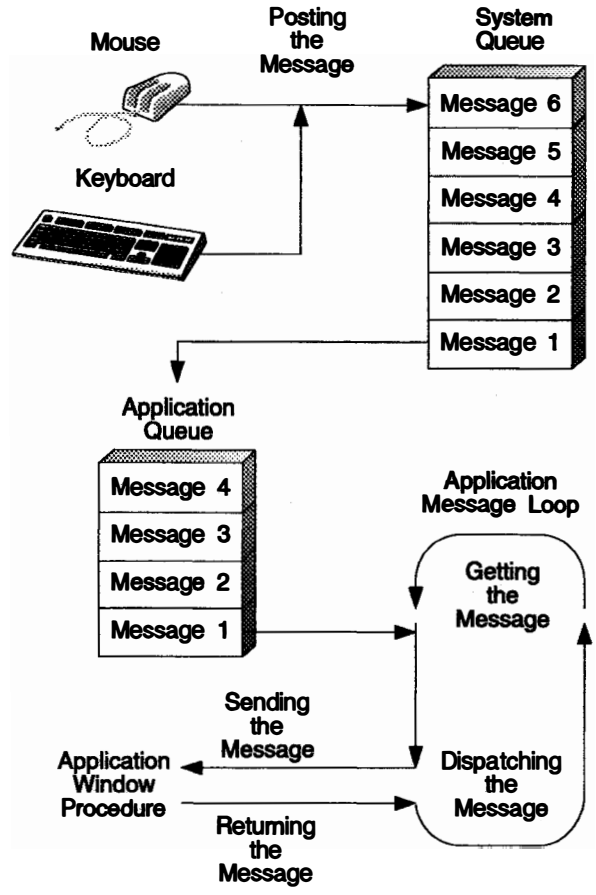



*Figure 1-5. Message-processing System*

In the PM environment, each input from the mouse or keyboard is delivered to an application as a message. A message cannot be processed before previous input, because the specific application and window for which input is intended are not known until all preceding input has been processed.

All input is first placed in a single queue called the *system queue*. The system queue, which is shared by all applications in the system, receives messages generated by the user from the mouse and keyboard. The system queue can hold the input from approximately 60 key presses and mouse clicks. The system queue can receive input from many sources, including the system itself, the timer, and other applications. However, only user input is processed synchronously.

The system queue temporarily stores user input so that nothing is lost if the user enters data faster than the application can process it. Generally, input is processed in the order in which it appears in the queue, but the application can change the order by *filtering* the input. Filtering is performed with API functions, but should be performed with discretion, because the processing of one input often changes the context for the next.

Each thread that receives input has an *application queue* allocated by an API function. This queue does not receive user input directly, but can receive other messages directly, such as messages from the system or timer. Messages are

removed from the application queue when the thread for which it is destined "gets" it. The messages are prioritized if more than one is waiting. If there are none, the thread is suspended until a message arrives.

The most efficient use of the system will be achieved if you structure your application so that one thread remains responsive to user input while others continue processing work. To be considered responsive to the user, the system must complete processing input within 0.5 seconds. That is, the thread handling input should check for the next message in the queue within that time.

A window procedure can control more than one window. The procedure receives messages in the form of four input parameters. The first parameter specifies the handle of the window for which the message is intended. The second parameter indicates the type of message. The last two parameters contain message parameters. Their interpretation depends on the particular message.

The window procedure processes the message, then sends a return value to the sending code. A window procedure must respond to all messages sent to it, even if the response is to send the message back to the system's default window procedure.

There are many types of messages, each with a unique identifier, and applications can define their own message types using a range of identifier values.

## Handling Application Resources

An *application resource* is held in a *resource file*, a file with information that helps to define a window. The resource file defines the names and attributes of the application resources that must be added to the application's executable file. The resources are:

**Accelerator table**
Used to define which key strokes are treated as *accelerators* and the commands into which they are translated. An accelerator is a single key stroke that invokes an application-defined function.

**Bit map**
A representation in memory of the data displayed on an all-points-addressable (APA) device, usually the screen.

**Dialog and window templates**
The definitions of a dialog box or window containing details of its position, appearance, its window identifier, and the identifiers of its child windows.

**Dialog include**
A definition of a dialog box in a header file.

**Fonts**
A typeface definition for character sets, marker sets, and pattern sets.

**Icon**
A graphical representation of an object, consisting of an image, image background, and a label.

**Menu**
A list of choices that can be applied to an object. A menu can contain choices that are not available for selection in certain contexts. Those choices are indicated by reduced contrast.

**Pointer**
The symbol displayed on the screen that can be moved by a pointing device. The pointer is defined in a bit map.

**String table**
A null-terminated ASCII string. A string table is loaded when needed by the executable file.

See the *OS/2 Programming Guide, Volume II* for guidance in creating a resource file. For guidance in building the resource file using the *Resource Compiler* (RC) utility in the IBM Developer's Toolkit for OS/2 2.0, see the online Tools Reference. The RC processes the resource text file to produce a binary file, then attaches it to the application's executable file so that an application can access its resources.

### Resource Editors

The *Dialog Box Editor* in the Toolkit enables you to design dialog boxes interactively on the screen and save the definitions in a resource file. The definition of the dialog box is included with other resource definitions in the application's resource file.

The *Font Editor* in the Toolkit enables you to edit font files interactively on the screen, save the definitions in a font file, and include the font file names in the application's resource file. The font file consists of a header file and a collection of character bit maps representing the individual letters, digits, and punctuation characters that display text on a screen.

The *Icon Editor* in the Toolkit enables you to create customized icons, pointers, and bit maps interactively on the screen and save the definitions in a resource file. You can work on a large-scale version of the icon or pointer while displaying a replica of the actual size.

## Exchanging Data Among Applications

Both users and active applications request an exchange of data with other applications. Data exchange requested by a user is held in an object called a *clipboard*. Requests by an application to exchange data with other applications is called *dynamic data exchange* (DDE).

### User-Generated Data Exchange

The user can transfer data from one application to another using the COPY, CUT, and PASTE commands. The first step is to copy or cut (delete) selected data from the source application; the data is now in the clipboard. Next, paste (insert) the clipboard data in the target application. The same process can be used to move data from one window to another within a single application. The CUT, COPY, and PASTE commands must be supported by an application as defined in CUA guidelines. They are implemented using a set of PM API functions.

The clipboard is the object that temporarily holds data. Generally, data is placed in the clipboard when a request to paste it is received. Once data has been sent to the clipboard, it should not be changed. Only one item of data at a time can be in the clipboard, but the data can be in a variety of formats, such as text, metafile, or bit map. The application can either define the formats or use one of the preregistered standard formats. The application also can register formats, as it can window classes, so they can be used by all applications in the system.

The application should support as many formats as possible to satisfy requests from target applications. For example, a spreadsheet application should support a spreadsheet format and as many common text formats as possible. Generating data

in all formats supported by an application can consume a lot of the operating system's resources. It would not make sense, for example, for a word-processing application to support a spreadsheet format because that format is beyond the scope of the operation of a word processor.

A clipboard can be owned by a thread. If a thread opens the clipboard, it has exclusive access to the clipboard until the thread closes the clipboard.

The clipboard is owned by the last window that requested ownership. Only the owning application can change the owner of the clipboard. If an owning window is canceled, data can remain in the clipboard. Before being canceled, the owning window must generate its data to satisfy subsequent paste requests.

### Application-Generated Data Exchange

The DDE protocol enables applications to access one another's data. Dynamic data exchange uses the PM message-processing system and the shared-memory interprocess communication protocol to pass large quantities of data among applications. All API functions that manage shared memory begin with the prefix Dos.

The data is held in a shared-memory segment that is available to designated applications or to all applications in the system. Data is passed in a mutually-agreed-upon format. An application that receives a handle to a data object in memory receives a message if the data changes. The application then can indicate, by way of a message, if it wants the changed data to be sent to it, or it can end the exchange. See "Shared Memory" on page 1-11 for information about shared memory.

### Direct Manipulation

Direct manipulation is a protocol that enables the user to visually drag an object in a window (the *source object*) and drop it on another object (the *target object*) in a window. This causes an interaction, or data exchange, between two windows.

The window containing the dragged object is referred to as the *source*. The window containing the object that was dropped on is referred to as the *target*. The source and the target can be the same window, different windows within the same application, or windows belonging to different applications. The dragged object can be the only visible object in the source window, or it can be one of many objects. The target object can be the only visible object in the target window, or it can be one of many objects.

A set of API functions with the prefix Drg supports requirements of source and target applications and are described in the *Presentation Manager Programming Reference, Volume I*.

## Information Presentation Facility

The Information Presentation Facility (IPF) is a tag language and compiler that supports the design and development of online information. Information can be associated with the windows and fields of an application, and can be displayed as a Help facility. Implementation of a Help facility requires two different development efforts:

- The development of the code that communicates with IPF and PM to display help panels.

- The development of a library of IPF tag-language files that define the contextual Help panels for the fields within the windows.

### Coding the Application

The PM code that calls the IPF Help hook is in the PM default window procedure. Use WinDefWindowProc for a standard window, or WinDefDlgProc for a dialog box. The functions relating to the creation of the Help instance are WinCreateHelpInstance, WinAssociateHelpInstance, and WinDestroyHelpInstance.

### Developing the Help Information

A Help panel is a unit of text within an application window. The database for the Help-panel text includes IPF tag language, which defines various characteristics of the text displayed in the Help text window. The IPF Compiler translates the tags into a format that is used by IPF. For more information about IPF, see the *Information Presentation Facility Guide and Reference*.

## Presentation Drivers

Presentation drivers are special-purpose I/O routines that handle field device-independent I/O requests from the PM and its applications. The requests are resolved with the device-specific attributes of the device. For detailed information about presentation drivers, see the *Presentation Driver Reference*.

The presentation driver is a higher-level interface to an output device than the OS/2 device driver. OS/2 device drivers are loaded at privilege level 0, the highest privilege level. All 16-bit presentation drivers are loaded at privilege level 2, while 32-bit presentation drivers are loaded at privilege level 3.

Presentation drivers are subsystems. They resolve I/O requests from PM and applications in the PM environment by interfacing directly with the device or with its device driver. Printer and plotter presentation drivers use the spooler interface beginning with the prefix Spl to pass data and controls to the OS/2 device drivers. Screen presentation drivers pass data and controls directly to the display hardware interface.

Presentation drivers are supplied as DLLs. When a PM program is initialized, the screen presentation driver is loaded and enabled automatically. Other presentation drivers, such as those supporting the printer and plotter, are loaded and enabled in response to the application requesting a device context.

## The Graphics Programming Interface

The graphics programming interface (GPI) consists of the OS/2 system functions that let you create device-independent graphics for your applications. The GPI functions are used in conjunction with the window manager to draw lines, shapes, and text in a window. Applications can also use the GPI functions to draw graphics output on such devices as raster printers and vector plotters.

## Presentation Spaces and Device Contexts

A presentation space is the key to an application's access to the system display, to printers, and to other graphics-output devices. Conceptually, a presentation space is a device-independent space in which you can create and manipulate graphics. The presentation space defines your drawing environment by specifying the tools

you have available to create graphics. These tools include the graphics primitives granted to every presentation space, in addition to the bit maps and fonts that your application loads for its exclusive use.

Actually, a presentation space is little more than a data structure whose fields contain values that define the drawing environment. The values represent the colors, widths, styles, and other attributes of the graphics you draw. The system creates the data structure when you create the presentation space and initializes the structure to default values.

You must create a presentation space to create graphics. You must also create a device context to display those graphics on a device. A device context is a bridge from a presentation space to a specific device. You create a device context by specifying the device you want to access and the type of access you want, such as direct or queued (for printing). You begin displaying graphics on the device by associating the device context with the presentation space. Once you have associated the device context, any lines, text, and images you draw in the presentation space are also displayed on the associated device.

Like a presentation space, a device context is a data structure. It contains information about the device driver that supports the specified device. The device driver interprets graphics commands sent to it from the presentation space and creates the corresponding commands for its device. It then sends the commands either directly to the device or to the spooler, depending on the type of access you gave the device context when you created it.

## Graphics Primitives

In OS/2 2.0, graphics primitives are lines, arcs, markers, text, and areas. They are called primitives because you use them as the basic tools to create the documents, pictures, and other composite graphics that your applications display to the user.

You draw a primitive by using a Gpi function. For example, to draw a line, you use the GpiLine function and specify the ending point of the line. The function uses the current point as the starting point for the line and draws from the starting point to the ending point. The current point is simply the ending point of the last primitive, unless you explicitly set the current point by using a function such as GpiMove.

A *line* primitive is a straight line. An *arc* primitive is a curve. Curves can be arcs of a circle or an ellipse, or they can be more complex curves, such as splines and fillets. A *marker* primitive is a mark or character that you draw at a specific point. Markers are typically used to plot points in a graph. An *area* primitive is a closed figure that can be filled with a pattern. A common use for an area primitive is to represent a cross-section in a mechanical drawing.

Every primitive has a corresponding set of primitive attributes. The attributes specify the color, style, size and orientation of the primitive when your application draws it. The primitive attributes are given default values when you create the presentation space, so you can use the primitives immediately. However, you can reset the attributes at any time. You have the choice of changing the attributes for individual primitives or changing a specific attribute for all primitives. For example, you can set the color for all primitives by using the GpiSetColor function, or you can set it just for the line primitive by using the GpiSetAttrs function.

## Graphics Objects and Operations

In addition to the graphics primitives, OS/2 2.0 provides graphics objects and operations that are used to manipulate primitives. Graphic objects are paths, bit maps, fonts, and logical color palettes. Graphic operations are clippings and transformations.

### Path

A *path* is a sequence of lines that you can use to create a filled area, a geometric line, or a clip path. A path is very much like an area primitive, in that you can use the path as a closed figure and fill it with a pattern. Unlike an area primitive, however, a path can be used to create geometric lines, sometimes called *wide lines*. Geometric lines are drawn, using a given width and pattern, so that they follow the outline specified by the path. The width of a geometric line can be transformed, unlike the width of a line primitive.

### Bit Map

A *bit map* is an array of bits that represents an image you can display on a raster output device. Bit maps typically represent scanned images and icons and are very much like image primitives, except a bit map can have several different formats, each specifying color information that an image primitive cannot contain. Also, bit maps can be used to create fill patterns to fill figures created by using paths and area primitives. Finally, bit maps can be copied from one presentation space to another or even from one location to another within the same presentation space.

### Font

A *font* is a collection of character primitives that share a common height, line weight, and appearance. The height of a font is specified in printer's points, a point being a typographic unit of measurement equal to 1/72 of an inch. A collection of fonts that share common stroke-width and serif characteristics is a font family. The term "stroke-width" refers to the width of lines used to draw characters and symbols from a font. A serif is a short cross-line drawn at the ends of the main strokes that form a character or symbol. Some common fonts are 12-point Helvetica**, 10-point Times Roman Bold, and 12-point Courier** Italic.

To use fonts in an application, first create a logical font. A logical font can be used only by the applications that defines it, and is lost when the presentation space is deleted. The logical font describes the typeface and other characteristics of the font and then assigns the font to the application's presentation space by passing to it a unique local identifier. (The identifier is a number in the range 1 through 254.)

### Logical Color Palette

A *logical color palette* is an array of colors that an application uses when drawing graphics. Any primitive or other graphic you draw has one of the colors given in the table. You specify the color by giving a *color index*. The index identifies the table entry, defining the color you want. Every presentation space has a default color palette when it is created, but you can create a logical color palette to replace it if you need other colors. Creating a new palette associates the color indexes with whatever colors you have specified in the corresponding palette entry.

### Clipping

*Clipping* is a process that limits graphics output to a specific region on the display screen or on a page of printer paper. You can use clipping with a presentation space by creating a *clipping area*. The clipping area is the region where the output is to appear. You can create this area by setting the dimensions of the graphics field and viewing limits, or by creating a *clip path* or *clip region*. A clip path is used to define a curved clipping area in world coordinates. A clip region is useful when an application must clip an area shaped like a rectangle or like a number of intersecting rectangles. If an application tries to draw output outside the clipping area, the system will "clip" the output, preventing it from appearing on the drawing surface of the output device.

### Transformation

A *transformation* defines how the system should map the points of one *coordinate space* into another. A coordinate space is a two-dimensional set of points used to generate output on a video display or printer. Because all graphics primitives and other drawing objects use coordinate spaces, a transformation affects the way all graphics are drawn by your application. For example, you can use a transformation to move a figure from one place to another on the display screen or to rotate or adjust the size of the figure. Transformations typically are used to give the user different perspectives of a single drawing, or to create rotated or sheared figures that would be time-consuming for the application to plot and draw.

## Drawing

You draw graphics by using the OS/2 drawing functions. A drawing function draws a primitive or other graphic, applying the primitive attributes that were current when the primitive was defined. For example, if the line color is changed between the defining of the two lines, each line will be drawn in its own color. Lines, for example, also have a line style attribute. The style determines whether the line is solid or a series of dashes, dots, or both.

Some attributes apply to all graphics primitives. For example, the foreground and background colors and mix modes affect all primitives. The foreground color defines the color of the primitive and the background color defines the color "behind" the primitive. For a line drawn with a dashed style, the dashes have the foreground color, and the gaps between the dashes have the background color. The mix modes define how the foreground and background colors are combined with colors already present. The mix mode can cause the color to overpaint the existing color, ignore it, or mix it by using a binary operator, such as the exclusive-OR operator.

Some attributes are specific to a particular graphics primitive. For example, the arc parameters apply only to arcs. The arc parameters specify a transformation that maps a circle to another circle, ellipse, or similar shape. When you draw an arc, the system uses the shape defined by the transformed circle as the shape for your arc. You supply a multiplier to set the final size of the arc.

A number of drawing functions use loadable resources to draw graphics. For example, the text-drawing functions, such as GpiCharString and GpiCharStringPos, can use a loaded font to draw text. To make a loadable resource available for these functions, you typically load the resource into memory and create a local identifier for the resource. For example, to use a font resource, you load it by using the

GpiLoadFonts function, and then set the local identifier with the GpiCreateLogFont function. Once you have a local identifier, you can set the resource to be the current resource by using a function such as GpiSetCharSet. Like the text-drawing functions, the marker and area functions can use resources when they draw.

## Retained Graphics and Segments

OS/2 2.0 enables you to retain the graphics you draw in your application by storing them in *retained segments*. You create a retained segment by setting the drawing mode of the presentation space to DM_RETAIN or DM_DRAWANDRETAIN and opening the segment. All subsequent graphics are stored in the segment (and are also drawn on the device, if you specified DM_DRAWANDRETAIN). You can close the segment at any time and draw the contents by using a function such as GpiDrawSegment. Retained segments are useful for storing graphics that the user provides. Once stored, the graphics can be redrawn or edited at any time. An element pointer enables the application to move to a specific graphics element in a segment. The element can then be drawn or replaced, or new elements can be inserted.

## Metafiles

A metafile is a file in which graphics data is stored. It is created with a *device context*. A device context links presentations to devices by converting device-independent presentation-space information into device-dependent information. A device context also gives applications access to important device information such as screen dimensions or printer capabilities. You associate the metafile device context with the presentation space, draw the graphics you want in the metafile, and then disassociate the device context from the presentation space and close it. Closing the metafile returns a handle that you can use to save the metafile in a disk file.

Metafiles are useful when transferring graphics images from one computer to another. An application can load a metafile from disk and transfer it into the application's presentation space. The presentation space can be associated with any device—display or printer. The graphics in the metafile are stored as graphics commands, not as a bit map, so an application can examine and extract portions of the metafile if necessary.

## Producing Hard-Copy Output

An application can send alphanumeric and graphics data directly to a hard-copy device (printer or plotter), or to the *spooler* for printing when a suitable device is free. The spooler creates print jobs and stores each as a temporary spool file on disk. The advantages of using the spooler are that it allows hard-copy devices to be shared among a number of processes in the system without interference. In addition, the spooler allows relatively slow printing operations to take place in the background, letting foreground applications continue to interact with the user. See "Presentation Drivers" on page 1-24 for more information.

There are two main ways to send data to a hard-copy device. The recommended way is to use the device functions to create a device context to which the data is sent. This method enables the application to control a print job. For example, it can choose the spool queue to which the data is sent and a priority for the job, and it can query and subsequently use special features of the output device that is selected by the spooler. For graphics applications, the graphics presentation space in which the data is drawn must be associated with the printer device context.

An application can request *direct* printing (spooler not used) when it creates the printer device context. Although this is not the recommended method, it enables specialized applications to use dedicated hard-copy devices that need not be shared with other applications running in the system.

An alternative method is to use the DOS functions. With this method, data is written to a spool file, if the spooler is running, or sent directly to the chosen printer if the spooler is not running. (Naturally, the data must be formatted for the print device.) However, the application has no control over the print job, so this method is not suitable for graphics applications.

The OS/2 2.0 spooler provides improvements in system use and performance. The new Spl functions improve access to spooler facilities. Spl functions control the PM print-spooling system, both locally and on a network. See the *Presentation Manager Programming Reference, Volume I* for a description of these functions.

## The OS/2 Application Programming Interface Functions

The OS/2 system functions give applications access to all the features of the operating system. These features, such as windows, device-independent graphics, and multitasking, enables you to create applications that make optimal use of the computer's memory, display, and processor while still meeting the needs of a wide range of users through either the traditional character-based interface or the PM graphical user interface.

The OS/2 system functions are organized into the following distinct groups:

*Table 1-1 (Page 1 of 2). OS/2 Function Groups*

| Group | Usage |
|---|---|
| Ddf | Dynamic-data formatting functions. Use to create and manage online, context-sensitive help information. These functions let you display both text and graphics and set up hypertext links between information units. |
| Dev | PM device functions. Use to open and control PM device drivers. These functions let you create device contexts that you can associate with a presentation space and use with the Gpi functions to carry device-independent graphics operations for displays, printers, and plotters. |
| Dos | Control Program functions. Use in full-screen and Presentation Manager sessions to read from and write to disk files, to allocate memory, to start threads and processes, to communicate with other processes, and to access computer devices directly. Most functions in this group can be used in PM applications. |
| Drg | Direct manipulation functions. Use to move graphical representations (OS/2 icons, for example) around the screen using a pointing device, such as a mouse. Drg functions let you initialize the structures that convey the necessary information about each object to the target and which describe the image to be displayed during the drag operation. They provide the system with the type, rendering mechanism, suggested name, container or folder name, name, true type, and native rendering mechanism of the objects being manipulated. |

*Table 1-1 (Page 2 of 2). OS/2 Function Groups*

| Group | Usage |
|---|---|
| Gpi | Graphic-programming-interface functions. Use to create graphics output for a display, a printer, or other output devices. The Gpi functions give you a full range of graphic primitives, from lines to complex curves to bit maps. You choose the attributes for the primitives (such as color, line width, and pattern) and then draw lines, character, and shapes. The retained-graphics capability lets you save the drawings in segments and build complex pictures by drawing a chain of segments. |
| Prf | Profile functions. Use to tailor some of the aspects of the system, including the names of ports, printers, printer drivers, and queues. Prf functions also enable you to change the spooler path, screen colors, the default printer and queue, the program list, and application settings. |
| Spl | Spooler functions. Use to allow your applications to write data direct to a spool file. This means that data by-passes the presentation driver, so it must be in a format that the printer can understand. Your applications must format the data. |
| Win | Window-manager functions. Use to create and manage windows. PM applications use windows as the main interface with the user. Win functions let you create menus, scroll bars, and dialog boxes that let the user select commands and supply input. Your application receives all mouse and keyboard input as messages from the message queue. Win functions let you retrieve messages from the queue and dispatch them to the window for which the input is intended. |

From the application's perspective, the code for each API function looks like a subroutine that can be called by name using an intersegment near call. The subroutine that implements the API call is not part of the application's executable file. Relocation pointers to the subroutines are contained in a *dynamic link library* (DLL), which is described in Chapter 5, "Dynamic Linking" on page 5-1. The loader fills in the addresses of the subroutines and prepares the executable for execution.

Before making an API call, the application pushes parameters onto an execution stack. After control is returned to the calling application, any error code is in the EAX register. By contrast, a PM application is returned data immediately in the EAX register, if the result of the call is a doubleword.

The call-return method of accessing system services has the following advantages:

- The application can call functions directly, eliminating the need for a DOS-style function router.
- The application's executable file is smaller than it would be if it contained all the code to implement the API calls.
- The API can be changed easily without affecting existing applications.

The implication is that the API can be extended with a new set of API functions. In fact, whether the kernel controls a function, or it is contained in a DLL, the mechanics of the API are the same. Following certain conventions, a developer can extend or replace an I/O subsystem.

The *OS/2 2.0 Control Program Programming Reference* and *Presentation Manager Programming Reference* contain a comprehensive, alphabetical listing of all the API functions, including their input and output parameters, error return codes, data structures, and messages.

# Chapter 2. The 32-bit OS/2 Programming Environment

This chapter describes:

- The 80386 Architecture
- How OS/2 2.0 takes advantage of the 80386
  - The Flat Memory Model
  - The Process Address Space
  - Memory Objects
  - Paging
  - Compatibility with 16-bit OS/2

## Intel 80386 Architecture

The Intel** 80386 is a powerful 32-bit microprocessor that forms the basis for OS/2 2.0. The 80386 incorporates multitasking support, sophisticated memory management, pipelined architecture, address translation caching, and a high-speed bus interface, all combined within the processor chip. While the 80386 represents a significant improvement over previous generations of Intel microprocessors, it retains software compatibility with older 16-bit microprocessors such as the 8086 and 80286 families.

The capacity of the 80386 processor is substantial. Some figures are presented below, in comparison with 80286 processors:

- 4, 5 or 6 million instructions per second (MIPS)[1].
- 4 Gigabyte physical address space, compared with the maximum of 16 Megabytes available on the 80286.
- 64 Terabyte virtual address space, compared with the 1 Gigabyte available on the 80286.
- Ability to handle memory objects from 1 byte to 4 Gigabytes in size, compared to segments of 16 bytes to 64 Kilobytes on the 80286.
- Paged memory management using 4 Kilobyte pages, compared to the 80286 which offered only segmented memory management.

This chapter provides an overview of the 80386 processor architecture, in order to serve as a base for understanding the changes made in OS/2 2.0. More detailed information about the 80386 can be found in:

- *Intel 80386 Hardware Reference Manual* (ISBN 1-55512-069-5)
- *Intel 386 DX Programmer's Guide* (ISBN 1-55512-082-2)
- *IBM PS/2 Model 80 Technical Reference* (IBM P/N 84X1508).

1 Sustained rate at clock speeds of 16, 20 and 25 MHz.

## Physical Characteristics

The 80386 processor consists of six dedicated units:

- Bus Interface Unit
- Code Prefetch Unit
- Instruction Decode Unit
- Execution Unit
- Segmentation Unit
- Paging Unit

These individual units are connected by 32-bit buses and operate in parallel to provide a 6-stage pipelined execution of instructions. This implies that up to six different instructions may be held concurrently within the chip, at different stages of execution. To further improve performance, the 80386 uses on-chip caching and implements sophisticated memory management and bit manipulation (such as a 64-bit barrel shifter) in the hardware.

The 80386 chip contains eight 32-bit general registers. To provide compatibility with the 8086 and 80286 processors, the 80386 provides the capability to use the lower-order 16 bits of these registers, or the upper and lower 8 bits of these registers, to represent the 16-bit registers used in these previous processors. This is illustrated in Figure 2-1.

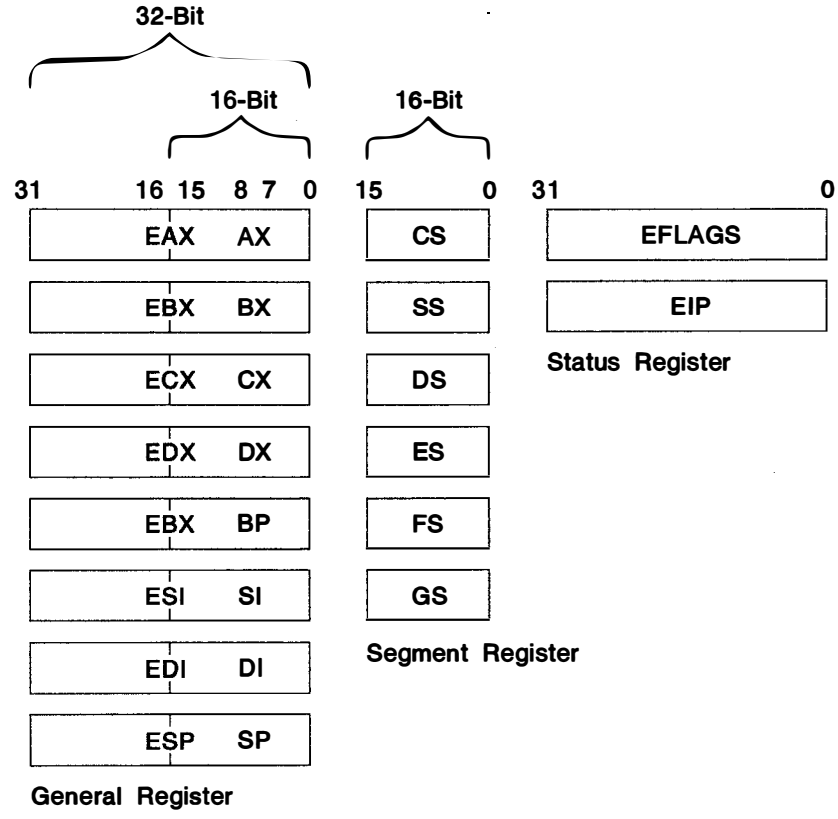



*Figure 2-1. 80386 General, Segment and Status Registers*

Note that programs running in virtual 8086 mode may utilize the full register set of the 80386 (all 32-bit registers including the new FS, GS, debug, control and test registers). The programs can also use instructions with 32-bit operands through the use of the size prefix.

The 80386 also provides six 16-bit segment registers which are used to contain segment selectors, thus enabling the same segmented memory model used in the 80286 processor. Note that the FS and GS segment registers are new in the 80386.

The instruction pointer (EIP) register and the flags (EFLAGS) register are both 32-bit registers.

Note that applications written for the 80286 run unmodified on the 80386. This is because the 80286 instructions, addresses, limits, segment types etc, are a full subset of those available in 80386, and run in 16-bit mode automatically. The 80386 handles this very simply; if the upper word (16 bits) of a memory reference is zero, then that reference must be an 80286 reference.

The registers described above are available to application programmers, either directly using assembly language or indirectly through the use of higher-level programming languages. The 80386 processor provides a number of additional registers which are not normally available to applications, but may be used by operating systems and system software.

- The 80386 provides four memory management registers:
  - Global Descriptor Table Register (GDTR)
  - Local Descriptor Table Register (LDTR)
  - Interrupt Descriptor Table Register (IDTR)
  - Task Register (TR)

  These registers contain pointers to data structures used by the segmented memory model, allowing compatibility with software written for the 80286 processor.
- Four control registers (CR0 to CR3) are used to contain pointers to data structures used by the paged memory model and for status information. These registers are new in the 80386 processor, since previous processors did not support the paged memory model.
- Eight debug registers (DR0 to DR7) and two test registers (TR6 and TR7) are provided to aid in real-time system and application debugging. These registers are also new in the 80386.

## Memory Addressing

The 80386 processor, like its predecessor the 80286, can operate in two addressing modes; *real mode* or *protect mode*. The memory addressing schemes used in each of these modes are described in the following sections.

### Real Mode

When the 80386 is powered up or re-initialized via a hardware reset, the processor is set into real mode. In real mode, the 80386 effectively operates as a 16-bit processor. Program addresses correspond directly to physical memory addresses. Memory is addressed using the segmented memory model only (paging is not supported), and the system's physical address space is limited to 1MB of real memory. Virtual memory is not supported in real mode.

The use of physical memory addresses directly by applications prevents any protection being applied by the processor to memory references. Hence applications executing in real mode may access one another's memory, or that of the operating system.

Segment registers are used to supply the base address for each type of memory segment (DS - data segment, CS - code segment, SS - stack segment and ES - extra segment). Figure 2-2 on page 2-4 shows how a segment is addressed in real mode.

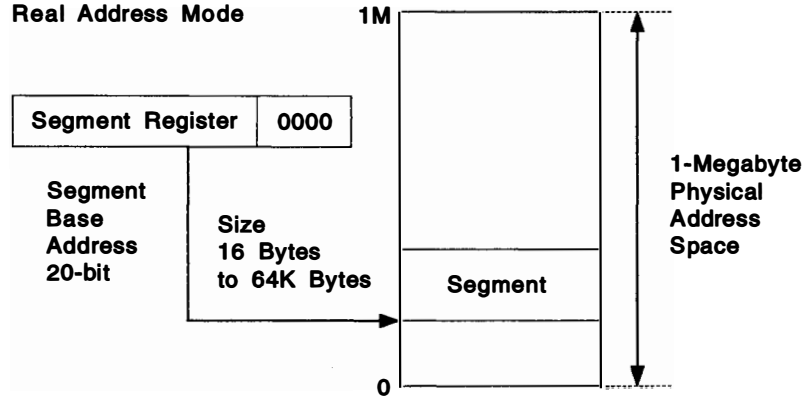



*Figure 2-2. Real Mode Addressing*

Each memory reference consists of a 16-bit *segment address* and a 16-bit *offset*. The processor automatically adds four binary zeros to the segment selector value (equivalent to multiplying by 16) to obtain a segment base address in memory. Thus a segment may start on any 16-byte boundary within the 1MB physical address space.

The required memory location within the segment is determined by adding the offset to the segment base address. Since the offset is 16 bits in length, the maximum offset (and therefore the maximum size of a segment) is 64KB.

## Protect Mode (Segmented Memory Model)

When the 80386 is switched to protect mode by software command, the full 32-bit capabilities of the processor are enabled, and the system's physical address space is increased to 4GB. Since virtual memory support is enabled in protect mode, the virtual address space visible to an application increases to a theoretical maximum of 64 Terabytes[2]. Each process occupies a separate logical address space, and the 80386 provides full memory protection between the address spaces of different processes, thereby preventing an application from inadvertently accessing and/or corrupting memory used by another application. Note however that under OS/2 2.0, multiple threads may be created within a single process, and dispatched independently by the operating system. These threads share a common address space, and it is therefore the responsibility of the application developer to ensure correct behavior of and synchronization between multiple threads within a single process.

In protect mode, the full 32 bits of the segment registers are used. These no longer contain the segment base address; rather, they contain a value which indexes a *descriptor table*, which in turn refers to the physical memory location of the segments. The lower-order 16 bits of the register contain the *segment selector*

[2] Note that in OS/2 2.0, this is is limited to 512MB per process in order to reserve memory for operating system use and to retain full compatibility with applications written for previous versions of the operating system, which used 16-bit addressing.

which identifies the segment, and the higher-order 16 bits are used internally to maintain control information about the segment.

Of the 16 bits which make up the segment selector, two bits are used to specify the privilege level of the segment, and one bit is used to select between the *Global Descriptor Table* (GDT) and a *Local Descriptor Table* (LDT). The GDT is used by the operating system or privileged software to maintain control over all segments within the system. A unique LDT is maintained for each process and used to control only the memory segments used by that process. In this way, each process is prevented from accessing the memory used by another process.

The remaining 13 bits are used as an index into the appropriate descriptor table, where the physical memory address of the segment is stored. This results in a total of 8192 entries per descriptor table.

A memory address is made up of the segment selector plus a 32-bit offset, which is added to the segment base address determined from the descriptor table, in order to produce a 32-bit linear address. This address translation operation is shown in Figure 2-3.

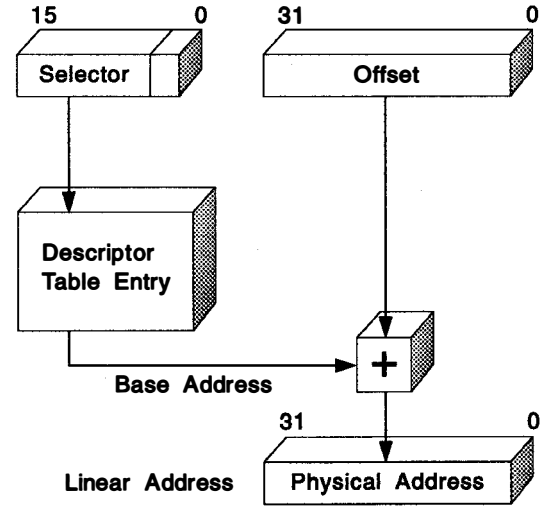



*Figure 2-3. Protect Mode Addressing—Without Paging*

The maximum allowable value for the offset, and thus the maximum size of a segment, is defined by two things. Each entry in a descriptor table contains a 20-bit limit field. These 20 bits allow a maximum segment size of 1MB, using the byte as the unit of size.

However, the descriptor table entry also contains a *granularity bit*, which specifies that either the *byte* or the *page* may be used as the unit of size in the limit field. When using page granularity, the 20 bits in the limit field represent a multiple of 4KB, allowing a segment size of 4KB to 4GB. Since the GDT (which contains *all* segments in the system) may contain up to 8192 entries, this results in a total system virtual address space of 64 Terabytes.

Note however, that OS/2 2.0 *does not* use the 80386 segmented memory model. Rather, it uses the 32-bit flat memory model, which results in a system global address space of 4GB. This reduces the complexities of application programming that are inherent in the segmented memory model, and increases the potential for portability of the operating system and application code to other hardware platforms. The maximum memory accessible by an application in the system, known as the process address space, is 512MB, which allows full compatibility with applications developed for previous versions of the operating system.

### Protect Mode (Flat Memory Model)

The 80386 is able to address very large memory (up to 4GB) in a single segment. It may therefore be desirable *not* to use the normal segmented memory model, but simply to use the entire system memory as a single linear address range. While the 80386 does not have a mode bit for disabling segmentation, the same effect can be achieved by mapping the stack, code, and data spaces to a single range of linear addresses. When this is done, the 32-bit offsets used by 80386 memory references can cover the entire linear address space.

OS/2 2.0 uses this technique to implement a flat addressing model. The operating system effectively creates a single code segment and a single data segment, with the base address of each segment selector set to zero (or any other address designated as a "relative" zero address), and a segment size of 4GB. This is equivalent to setting the same segment selector into the CS, DS, ES, FS, and SS registers. The selectors for these segments are allocated within the GDT, and the offsets are therefore equivalent to linear memory addresses.

The advantage of using such a technique is that it greatly simplifies memory management within an application, since the application developer no longer need be concerned with the internal implementation of data structures as segments with a defined maximum size. The use of a flat memory model also facilitates migration of the operating system and application code to other hardware platforms, since the code is not explicitly designed around the segmented memory model.

## Paging

In addition to the segmented memory management offered on the earlier 80286 processors, the 80386 provides a paged memory model. This is an optional function of the 80386, and there are no direct performance implications if an operating system chooses not to use paged memory. However, the paged memory model provides significant performance benefits when running large applications which make extensive use of virtual memory.

Under previous versions of the operating system, the smallest unit of memory (for memory management purposes) was the segment, since the operating system was designed to execute on the 80286 processor and use the segmented memory model. With the 80286, segments may vary in size between 16 bytes and 64KB; therefore, there is a danger of having a large amount of free memory which is fragmented into small, discontiguous units.

Previous versions of OS/2 manage this by moving segments within real storage to create a larger free space, and by swapping unused segments to disk until they are required. This entails a high degree of overhead for the operating system. With an 80386 processor however, segments may be up to 4GB in size, and the overhead can potentially result in an unacceptable performance impact, particularly for applications with very large segments.

To avoid this situation, the 80386 processor provides a paged memory model, implemented in hardware through a dedicated paging unit on the chip. A page is a 4KB unit of contiguous memory, and replaces the segment as the unit of granularity for memory management, including swapping to and from disk. Note that paging is available in protect mode, in conjunction with both the segmented and flat memory models.

Using the paged memory model, an application makes a memory reference in the normal way, using either the segmented memory model or the flat memory model. In the case of a segmented memory model, the segmentation unit in the processor automatically resolves the reference into a 32-bit value. However, this does not represent a linear address, but is comprised as follows:

- The high-order 10 bits of the field are used as an index into a Page Directory table. The entry in this table in turn refers to the base address of a Page Table.
- The next 10 bits of the field are used as an index into the Page Table referred to by the Page Directory entry. The entry in the Page Table provides the physical base address of a 4KB page.
- The lower-order 12 bits of the field are used as an offset within the page referred to by the Page Table entry.

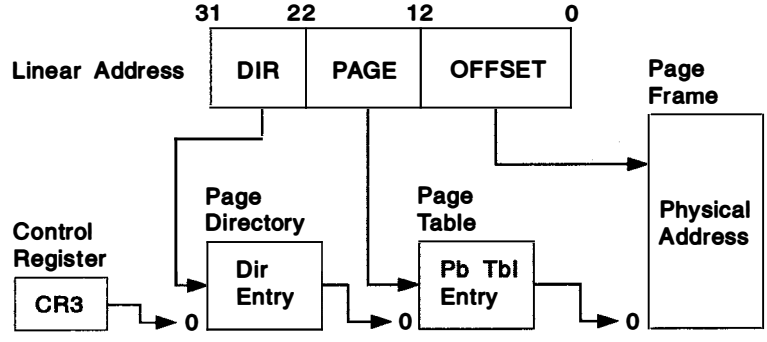



*Figure 2-4. Protect Mode Addressing—With Paging*

Both the Page Directory and Page Tables contain 32-bit page specifiers. The Page Directory and Page Tables are themselves contained within single pages, and may therefore contain a maximum of 1024 entries. Each page directory can therefore access up to 4GB of storage, which is the maximum physical address space of the 80386.

Pages may be shared between processes by defining them in the Page Tables of each process. Note that this is done at the Page Table level rather than the Page Directory level, in order to share *only* the individual pages required.

In order to further reduce the overhead involved in looking up page references, the 80386 also provides a hardware-based address caching mechanism for paging information. This is known as the Translation Lookaside Buffer (TLB). The TLB contains the physical addresses for the 32 most-recently-used pages, and therefore allows very fast access to these pages since it bypasses the normal page translation process. Use of the TLB is handled entirely within the paging unit, and is not visible to software.

# Protection

The 80386 processor implements five different types of protection for tasks executing within the system. These are:

- Type checking
- Limit checking
- Privilege levels
- Restriction of procedure entry points
- Restriction of instruction set

Each instruction and memory reference is checked by the hardware to ensure compliance with the protection rules prior to execution. For memory references, checking is performed during the address translation process, and is applied to both segmented and paged memory models. Protection parameters are stored in the segment descriptors or in the Page Directory and Page Table entries.

## Type Checking

With each descriptor, there is a *type* field which is used to distinguish between the different descriptor formats. This field also specifies the intended use of a segment. For example, the allowable types for data segments are:

- Read-Only
- Read/Write

and for code segments:

- Execute-Only
- Execute/Read

The type field therefore ensures that segments are only used in ways for which they are intended. For example:

- The code segment (CS) register may only be loaded with the selector of an executable segment.
- Selectors of executable segments that are not defined as readable cannot be loaded into data segment (DS, ES, FS and GS) registers.
- No instruction may write into an executable segment.
- No instruction may write into a data segment unless that segment is defined as *Read/Write*.
- No instruction may read an executable segment unless that segment is defined as *Execute/Read*.

## Limit Checking

The limit field in each segment descriptor is used by the processor to prevent a program addressing memory outside the segment through the use of an overly large offset value. During address translation, the offset value specified in the memory reference is compared with the limit field, and an exception is generated if the offset is larger than the limit for that segment. The limit field in general prevents errors in one program from corrupting other programs' code or data areas.

## Privilege Levels

The 80386 implements a four-level protection mechanism. Level 0 is the most privileged, and Level 3 is the least privileged. The four levels may be visualized as concentric rings, with the most privileged level in the center. All code and data segments in the system are assigned a privilege level, which is stored in the segment descriptor. At any one moment, a task executes only on one of the four levels:

**Level 0** This is the most privileged level, and code executing on this level may use all protect mode processor instructions. This level is used by those routines in an operating system which are essential for resource allocation and control. This part of the system is often referred to as the kernel or nucleus.

**Level 1** This is the second most privileged level and is normally used for the remainder of the operating system routines and for the Input/Output support routines. Note that Level 1 is not used by OS/2 2.0.

**Level 2** This level is typically used as the application services level. It should be used for routines that do not belong to the operating system, but should still be protected from user code, such as communications support and database management programs.

**Level 3** This is the least privileged level and should be assigned to user application program code.

A task executing at one protection level cannot access data at a more privileged level (Level 3 cannot access data at Level 1), nor can it invoke a procedure at a less privileged level (Level 1 cannot invoke Level 3). Thus, both access to data and transfer of control are restricted in appropriate ways. The processor interprets the protection parameters and automatically performs all the checking necessary to implement this protection.

## Restriction of Procedure Entry Points

To achieve transfer of control between procedures on different privilege levels, a descriptor type called a *gate* is provided. Programs wishing to transfer control call the gate by specifying a segment base address, rather than transferring control directly to the required procedure.

The four types of gates are CALL gates, TASK gates, INTR (Interrupt) gates and TRAP gates. The routine invoked when the gate is called simply redirects control to a new address which contains the privileged routine to be executed.

From the program's point of view, this is no different from transferring control to another code segment, since the calling instruction simply regards the gate as another segment. However, it effectively isolates the calling procedure from the called procedure, and since only the base address is supplied in the calling instruction, the calling procedure has no access to any point other than the defined entry point of the called procedure.

Calls are verified to ensure that they satisfy two conditions:

1. The call must enter the called procedure at the beginning of that procedure; this is normally ensured by the gate descriptor itself, which supplies the necessary offset to the entry point.
2. The code segment for the called procedure must have the same privilege level as the gate descriptor.

### Reserved Instructions

Certain processor instructions are reserved for execution only by the operating system, and may therefore execute only at privilege level zero. Such instructions include HLT (Halt Processor), LGDT (Load GDT), and LTR (Load Task Register).

In addition, some I/O instructions are restricted:

1. The IOPL field in the EFLAGS register defines whether or not the current task has the right to use I/O-related instructions.
2. The I/O Permission Bit Map in the TSS determines whether the current task may use ports in the I/O address space.

## Interrupts

When the processor is running in protect mode, interrupts are not vectored from the base of memory. Instead, each interrupt has a code which is used as an index into an *Interrupt Descriptor Table* (IDT), the base address of which is contained in the Interrupt Descriptor Table Register. There may be up to 256 interrupt and exception codes, generated by devices or by software.

At system initialization, the IDT is loaded into memory by the operating system, and its location is stored in the IDT Register. Each descriptor in the IDT specifies the address of the interrupt handler routine which will service interrupts with that code.

There are three types of gate descriptors in the IDT:

- Interrupt Gate Descriptors
- Trap Gate Descriptors
- Task Gate Descriptors

For interrupt and trap gates, the descriptor in the IDT contains the selector of the gate, and therefore points indirectly to a procedure that will execute within the current task, since the selector within the gate procedure points directly to an executable segment descriptor in the GDT or the current LDT. This takes place exactly as if the 80386 was calling a procedure within the current application.

For the task gate however, the selector within the gate points to a TSS descriptor in the GDT. Invoking the task gate therefore causes a task switch to occur. There are certain advantages to the use of a task gate:, since it allows a program to pass control to a higher privilege level, and the application may therefore invoke operating system routines to process interrupts and exceptions. In addition, the new task may be given its own LDT to prevent it from accessing memory used by the current task, and the TSS of the current task is automatically saved.

However, there are also performance implications in using task switching. Interrupt handling through task switching requires approximately 15 microseconds on a 20 MHz 80386, while switching to a procedure within the current task takes about 3.6 microseconds. But the advantages of having the operating system manage exceptions (smaller application code, greater portability, standard exception handling) typically outweigh the slight performance penalty.

## Input/Output Processing

I/O addressing on the 80386 may be performed either by issuing specific I/O instructions to the I/O address space, or issuing general-purpose memory manipulation instructions to memory-mapped I/O.

The I/O address space is separate from the linear physical memory and the I/O instructions do not go through the segmentation or paging hardware. The I/O address space is 64KB in size. It may be mapped in various ways; for instance, 64KB of individually addressable 8-bit ports, 32KB of 16-bit ports, 16KB of 32-bit ports, or any combination of the above up to the maximum allowed 64KB. The processor can transfer 32 bits of data at a time to a device located in the I/O address space, using the IN, OUT, INS and OUTS commands.

The I/O address space has two protection mechanisms:

1. The I/O Privilege Level (IOPL) field in the EFLAGS register controls access to the I/O instructions.

   The IN, INS, OUT, OUTS, CLI and STI instructions are only allowed to execute if the CPL (Current Privilege Level in the CS descriptor for the active task) is less than or equal to the value of the IOPL field.

   Only system code (privilege level 0) can change the IOPL value.

2. The I/O permission bit map in the active TSS controls access to individual ports in the I/O address space.

   There is one bit for each 8-bit port in the I/O address space, which means that the I/O permission bit map could be up to 64 Kilobits (8 KB). If a task references an I/O port and the corresponding bit is on, the processor signals a general protection exception. The exception can then be handled by the system software to initiate an exception handling procedure within the current task, or to initiate a new task, which will redirect the I/O.

   By changing bits in the I/O permission bit map of different tasks' TSSs, an operating system can allocate ports to tasks and avoid having two tasks use the same port concurrently.

## Virtual 8086 Mode

The 80386 processor supports concurrent execution of one or more 8086 programs within the protect mode environment. There is no longer a need for the processor to switch back to real mode in order to simulate a 8086 machine.

An 8086 program runs in protect mode as part of a *virtual 8086 task*. Virtual 8086 tasks are able to take advantage of the 80386 hardware support for multitasking, offered in protect mode. Virtual 8086 tasks may execute concurrently with one another and with other protect mode tasks in the system.

The purpose of the virtual 8086 task is to form a *virtual machine* for running programs written for the 8086 processor. A complete virtual machine consists of the 80386 processor support, plus additional support from operating system software:

- The hardware provides a set of virtual registers (implemented through the TSS), a virtual memory space (the first 1MB of the 32-bit linear address space) and directly executes all instructions that deal with these registers and with this address space. Figure 2-5 on page 2-12 shows the way in which the memory used by virtual 8086 machines is mapped into the system's physical address space.

- The operating system software controls the external interfaces of the virtual machine (I/O, interrupts and exceptions). In the case of I/O, the operating system can choose either to emulate I/O instructions or to let the hardware execute them directly.

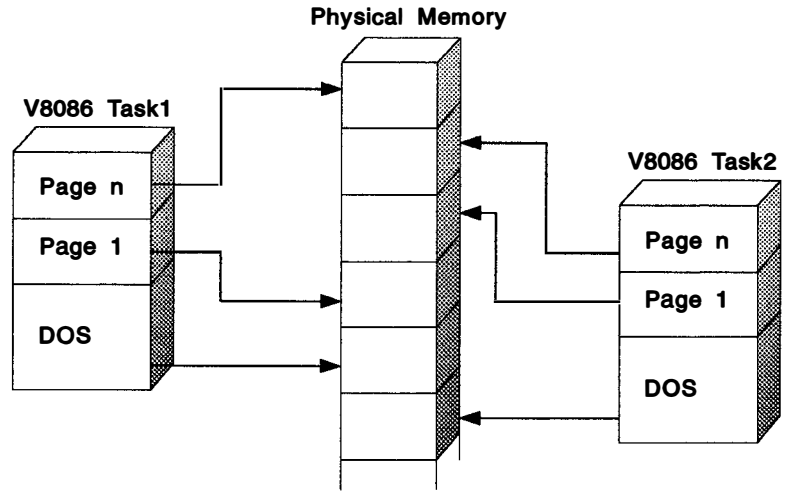



*Figure 2-5. Virtual 8086 Environment—Memory Management*

Virtual 8086 tasks execute at privilege level 3 (lowest) and are subject to all of the protection checks defined in protect mode, thereby preventing an ill-behaved application from accessing and potentially corrupting memory used by other tasks in the system.

All I/O is normally handled through the I/O Permission Map in the 80386 TSS for both virtual 8086 applications and other protect mode applications. This means that any call to I/O services generates an exception which is trapped by the 80386 and may then be handled by the operating system. Any unauthorized calls may be trapped within the operating system, thus preventing an ill-behaved application from "hanging" the system.

In addition, the 80386 paging hardware allows virtual 8086 tasks to share segments.

## Numeric Coprocessor

The 80386 processor may operate in conjunction with, and utilize the features of either the Intel 80287 or 80387 numeric coprocessors. When the system is initialized, the presence of a numeric coprocessor, and its type if present, is checked by the 80386. If an 80287 coprocessor is detected, the 80386 automatically converts all memory transfers to 16-bit format.[3] If an 80387 is detected, it is used in 32-bit mode, thereby utilizing the full potential of both the 80386 and 80387.

[3] Note that IBM does not support or recommend the use of 80287 numeric coprocessors in 80386-based systems. For a list of supported numeric coprocessors for each system unit, readers should refer to the appropriate IBM Product Announcement for that system unit.

## Coprocessing

Besides the Intel 80287 and 80387 numeric coprocessors, the 80386 supports multiple 80386 processors within the same system, sharing memory and other resources. This support is provided through the LOCK prefix instruction. When specified in conjunction with another instruction, the LOCK prefix instruction ensures that the locking processor has exclusive use of the requested resource.

Only a few 80386 instructions can be used with the LOCK prefix instruction. It is typically used to prefix instructions like BTC (Bit Test and Complement) where it locks the area of memory defined by the destination operand for as many cycles as necessary to update the entire operand.

In several instances, the processor itself automatically locks activities on the data bus. For example, when acknowledging interrupts, switching tasks, loading descriptors from the LDT to the segment selector and updating the page table ACCESS and DIRTY bits, the required memory pages are locked since these are highly critical operations.

The 80386 includes on-chip memory caching to improve performance. The processor must therefore allow for the case where data in shared memory is modified and where that data is currently recorded in a cache on another processor. In such situations, the 80386 employs an interprocess or interrupt to let other processors know when such a change has been made.

This is normally done by using one of the physical address pins on the chip, and having the receiving processor implement a task switch when it receives this signal. The task switch clears the system registers, reloads the new descriptors and invalidates the memory cache in the processor.

Note that by changing the function of one of the addressing pins however, the physical addressing capability of the processor is reduced to 2GB.

## OS/2 and the 80386 Processor

OS/2 2.0 is the first version of the OS/2 operating system to take advantage of the 32-bit features of the Intel 80386 microprocessor. Applications, subsystems, and the operating system can utilize the full register set of the 80386, 32-bit instructions and addressing modes, and memory objects larger than 64KB.

In OS/2 2.0, processes view memory as a large linear address space addressable by 32-bit offsets from the beginning of memory. That is why the OS/2 2.0 memory model is known as the "0:32 memory model," or the flat memory model. 16-bit OS/2 uses the 16:16 model for addressing memory, in which a segment and the offset into that segment must be specified in order to address a single byte of memory. The flat model effectively hides all segmentation from the 32-bit programmer, resulting in a portable programming model with much higher performance than a segmented system can provide.

The high performance of applications, subsystems, and the operating system using the flat model is derived from several areas. In the segmented or 16-bit model, segment registers have to be reloaded with selectors every time a different 64KB of memory needs to be accessed. These selector-load operations are very expensive in protect mode on 80X86 processors because of the checking that must occur to ensure segment protection. In the flat model, a 32-bit offset relative to the base of the linear address space is used to address any byte of memory without reloading

any selectors; 32-bit programs and subsystems do not use or know about segment registers.

Another benefit of the flat model is that there is only one memory model for applications (small with no 64KB restriction) instead of the many models (small, medium, large, huge) used in previous versions of the OS/2 operating system and all other systems that target 8088 and 80286 microprocessors.

## Process Address Space

In the 16-bit version of the operating system, a process address space consists of a collection of segments mapped by a Local Descriptor Table (LDT). A byte-addressable segment is the basic unit of allocation and sharing. In the 32-bit version of the operating system, a processes address space consists of a single, large ring 3 segment that the process addresses using the 32-bit offsets.

The size of the process address space in OS/2 2.0 is 512MB. Protected memory reduces this to 448MB. This restriction is required to maintain compatibility with the 16-bit version of the operating system. Figure 2-6 shows the process address space in the flat memory model.

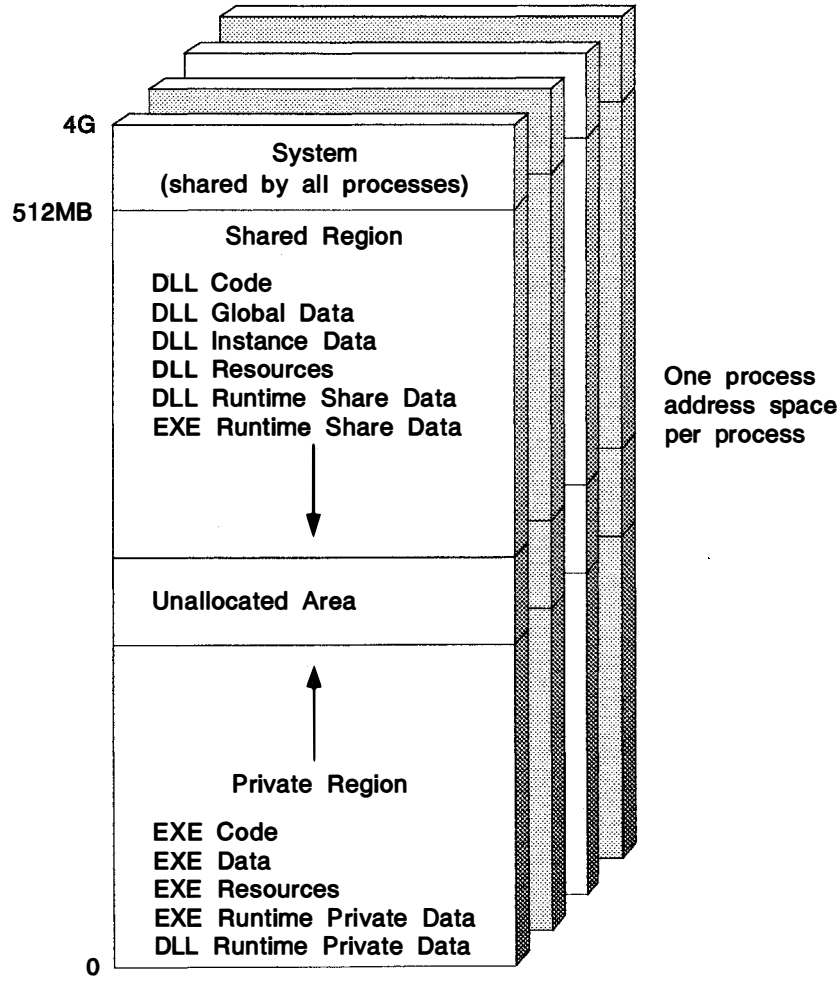



*Figure 2-6. OS/2 2.0 Process Address Space*

Notice in the Figure 2-6 that the system is at the top of each process address space. Anytime an application attempts to use an address larger than 512MB or an address that has not been allocated, an access fault is generated. The process address space itself is divided into private and shared regions, much like the LDTs in 16-bit OS/2. Shared memory allocations grow down from the top of the address space, whereas private allocations grow up from the bottom of the address space. While the shared and private address spaces grow toward each other, they are not allowed to overlap. Private and shared address spaces have a guaranteed minimum size of 64MB. This means that shared addresses may not be allocated within the first 64MB of the address space, and private addresses may not be allocated within the last 64MB of the 512MB address space.

## Memory Objects and Memory Sharing

In the flat memory model the unit of memory allocation and sharing is called a memory object. Memory objects are:

- not relocatable
- allocated in units of 4KB (page size)
- aligned in linear space on 4KB (page size) boundaries

Instead of being divided into segments as it is in 16-bit OS/2, memory is divided into memory objects that consist of one or more 4KB pages.

These characteristics imply that memory objects must be allocated with an upper bound specified for their size, and that memory allocation reserves this linear space for that object. The size of a memory object is defined when the object is created. After a memory object is created, its virtual address does not grow or shrink beyond its initially defined size.

Because sharing is done by sharing linear addresses, a shared memory object's linear address range is reserved in all process address spaces. Instance data is allocated in one of the shared address regions with private pages behind it. Instance data is treated as a memory object and is either page or page table aligned, depending on the performance characteristics required. Figure 2-7 on page 2-16 shows how memory objects are mapped into the process address spaces provided by the system and shows the relationship between private and shared regions.

Memory objects may not be allocated at specific addresses during runtime. However, both private and shared memory objects defined in an .EXE program module file are allocated in the private virtual address space and may be allocated at specific addresses that are defined in the .EXE file format. Because of the 64MB minimum private address space size, a program module may specify the addresses of its memory within this address space and be guaranteed the linear addresses will be available during program load. Specific allocations above the first 64MB may not succeed because the desired addresses may be occupied by shared memory objects.

The only way to guarantee a specific linear address range will be reserved for a program's use is to define a memory object in the program module .EXE file that spans the desired linear address range. As long as a program module only uses the address space below 64MB, the system will guarantee the ability to reserve the specific linear addresses using this mechanism. This feature does NOT apply to DLLs.

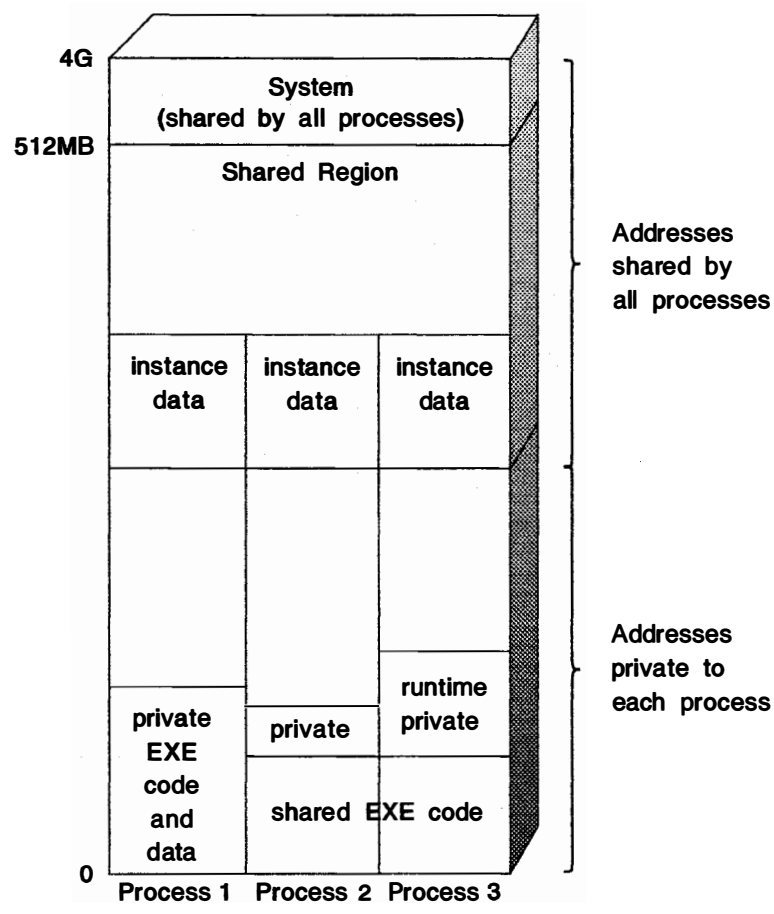



*Figure 2-7. Multiple Linear Address Space Management*

Reserved linear address ranges are invalid to address until memory has been committed for the address range. The commitment of memory to an address range makes the address range valid to access, the decommitment of memory from an address range makes the address range invalid to access. The reservation of a linear address range does not guarantee that memory resources will be available to commit to the reserved address range.

The backing storage for a range of pages may be committed and decommitted within a previously allocated private or shared memory object. This allows an application to make specific address ranges within an object valid or invalid as the application sees fit. Commitment and decommitment always take place on a page granularity. To assist an application or library in its management of committed memory, the system can provide the system page size through the DosQuerySysInfo function.

The operating system does not use the virtual address as the protection domain, and therefore never checks to ensure that an argument data structure resides within a single private or shared memory object. In general passing an argument that crosses multiple memory objects is considered poor behavior and should be avoided. Also the locking mechanism provided for device drivers for DMA does not

allow a device driver to lock memory that crosses multiple memory objects. Therefore even though the system does not prevent an argument from spanning multiple objects, an API call may fail if it does.

## Page Attributes and Memory Access Protection

Page-level memory protection is one of the significant differences between the 32-bit flat memory model and the 16-bit segmented model. In the 16-bit segmented model, protection exists on a per-segment basis. Access and limit checks occur on each segment selector load. Within the flat model, however, because an application's memory objects exist within the same segment, the Intel segment protection semantics are bypassed. Page-level protection is used to manage the memory within a process's address space, but only for DLLs.

When allocating memory using the flat model, an application can control the attributes of the pages within the range of addresses spanned by the allocated memory object. The commitment of memory to an address range may also be controlled by the application so that specific address ranges within a memory object can be made valid or invalid as the application sees fit. If an invalid page is addressed, an access fault occurs. Commitment, decommitment, and changing of the attributes of a page are done on a page granularity. To assist applications and libraries in their management of committed memory, OS/2 2.0 provides the system page size (4KB on an 80386). The following summarizes the attributes that can be associated with a single page or group of pages within a memory object:

| | |
|---|---|
| **COMMIT** | Page is committed with backing storage. |
| **DECOMMIT** | Page is not committed and access will cause an access fault |
| **EXECUTE** | Execute access to the committed page is allowed. |
| **READ** | Read access to the committed page is allowed. |
| **WRITE** | Write access to the committed page is allowed. |
| **GUARD** | The committed page is a guard page. |

On the 80386, execute and read access are equivalent, and write access implies both read and execute access. The guard page attribute is used to facilitate automatic stack growth and stack limit checking, but may be used by an application in other data structures where appropriate. Because a memory object can contain some committed and some decommitted pages, "sparse" memory objects can be used with the flat memory model.

While the paging feature of the 80386 is used to support the flat model and multiple DOS address spaces, it also allows OS/2 2.0 to provide a different kind of memory overcommitment than previous versions of the operating system. In 16-bit OS/2, segment swapping is used to keep the system running in memory-stressed conditions. Due to the I/O performance of most fixed disks, however, segment swapping does not perform well enough to provide generic virtual storage on demand. The 80386, in comparison, provides paging. From a system perspective, a lot of storage can be virtualized on fixed disk media at a much lower I/O cost because the size of a page is fixed and is smaller than that of a segment. The system also can do a better job of tracking memory usage because memory paging algorithms operate on a page granularity instead of a segment granularity. For these reasons, OS/2 2.0 is a demand-paged virtual storage system which is designed so that the system runs acceptably in overcommitted situations.

## Compatibility with 16-Bit OS/2

OS/2 2.0 runs all 16-bit OS/2 applications and subsystems. To do this, 16-bit and 32-bit modules reside simultaneously. For 16-bit and 32-bit modules to coexist, however, memory must be addressable from each model. This means that a high-performance mechanism for converting 16-bit addresses to 32-bit addresses and vice versa is needed. The technique used to deal with address conversions between the segmented and flat memory models is called *LDT tiling*.

A tiled LDT contains at most 8192 descriptors where each descriptor maps a 64K region of the process address space and the regions mapped by these descriptors are contiguous in the linear address space. The result is that a single LDT can map, at most, 512MB of contiguous linear address space in the flat model. (This is why OS/2 2.0 restricts the size of the process address space to 512MB.)

This tiling of the LDT creates an address mapping between the 16:16 address and the 0:32 address for any byte of memory in the process address space. Figure 2-8 shows how 16-bit segments are allocated in the process address space and mapped into the LDT.

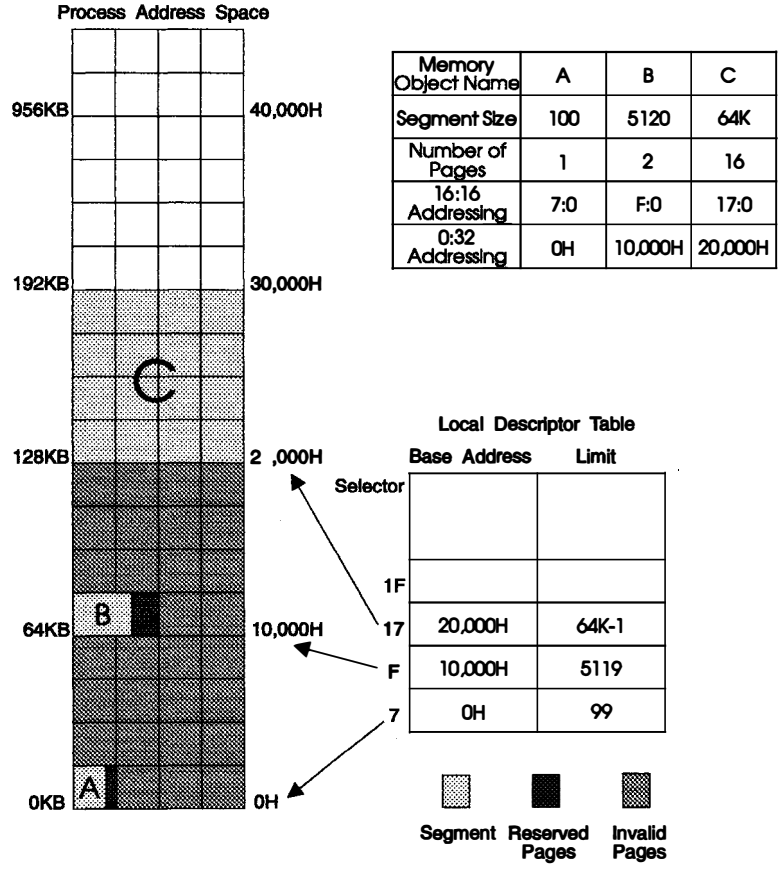



*Figure 2-8. Tiled Address Space Next to LDT*

To make this address mapping work, the LDT is managed differently in 32-bit OS/2 than in 16-bit OS/2. In 16-bit OS/2 the LDT selectors for private and shared memory are interspersed in the LDT. The LDT in 32-bit OS/2 is managed sparsely; private selectors are allocated from one end of the LDT while shared selectors are allocated from the other. This does not affect the compatibility of 16-bit modules, because their segmented nature makes them relocatable by default.

The memory manager puts all 16-bit API memory requests on 64KB linear boundaries and sets up the LDT to access the memory as described. This guarantees that the system can satisfy 16-bit DosReallocSeg requests up to the 64KB maximum. Notice in Figure 2-8 that each 16-bit segment takes up at least one page of virtual and physical memory, as well as one page on the swap device. Therefore, 16-bit applications that create many small segments will fragment the pages that constitute the address space. More memory is consumed than when the same application is run on 16-bit OS/2, although part of this memory fragmentation effect is reduced because the system swaps pages instead of segments. The trade-off is necessary, however, to provide a high-performance mechanism for address sharing between the two different memory models.

The following API functions are used to convert addresses between the 16-bit and 32-bit models:

- DosSelToFlat
- DosFlatToSel

  **Note:** See "DosSelToFlat" on page 3-17 and "DosFlatToSel" on page 3-17 for more information on using these API functions.

LDT tiling also provides a mechanism for 32-bit applications to utilize 16-bit DLLs. 32-bit modules can quickly create 16:16 aliases for memory objects. Because the aliasing is formula-based, a mapping layer can be provided for 16-bit DLLs. A function that performs this mapping is called a "thunk."

## Summary

The 80386 is a powerful 32-bit microprocessor that forms the basis of OS/2 2.0. It consists of six dedicated units connected by 32-bit buses operating in parallel. This enables up to six different instructions to be held concurrently within the chip. In addition, the 80386 processor contains eight 32-bit registers (which are compatible with 8086 and 80286 processors) and six 16-bit registers. Other features of the 80386 processor include:

- The ability to operate in two addressing modes: *real* and *protect*. In real mode, the 80386 operates as a 16-bit processor. In protect mode, the processor can take advantage of 32-bit capabilities, such as the ability to address very large memory in a single segment and to manage memory as pages, not as segments.
- The ability to support a multitasking environment.
- A greater degree of protection for tasks executing within the system. The types of protection are type checking, limit checking, privilege levels, restriction of procedure entry points, and restriction of the instruction set.
- Improved interrupt handling.
- The ability to perform I/O addressing either by issuing specific I/O instructions to the I/O space, or issuing general-purpose memory manipulation instructions to memory-mapped I/O.
- Support for concurrent execution of one or more 8086 programs within the protect mode environment.
- The ability to operate in conjunction with, and utilize the features of either the Intel 80287 or 80387 numeric coprocessor.

OS/2 2.0 takes advantage of the 32-bit features of the 80386 processor. It uses the 32-bit instruction and addressing modes of the 80386 processor to implement a *flat memory model* The flat memory model provides the following benefits:

- A larger process space (512MB)
- Division of memory into *memory objects* instead of segments. Memory objects consist of one or more 4KB pages.
- Page-level protection.

OS/2 2.0 maintains compatibility with previous version of the operating system. This makes it possible for 16-bit and 32-bit modules to coexist; however, memory must be addressable from each model. The technique used to convert between 16-bit and 32-bit addresses is called *LDT tiling*.

# Chapter 3. The Application Development Environment

This chapter discusses:

- The types of applications that can run under OS/2 2.0
- The programming models for OS/2 2.0 applications
- The program development environment for OS/2 2.0
- The issues the programmer faces in migrating to the 32-bit OS/2 programming model

## Applications Running Under OS/2

Two forms of input and output are supported with 32-bit functions: the C interface (producing command-line-based applications that can be run in a text window or full screen) and PM applications. Applications that must create graphics, handle keyboard or mouse input, or intercept or modify device information should do so within the context of the PM functions. The 32-bit PM programming model is the preferred programming model for OS/2 2.0 applications. Although OS/2 2.0 can execute any application developed for version 1.X of the operating system, only 32-bit PM applications can take full advantage of the graphical user interface and the 32-bit features of OS/2 2.0. The types of applications supported by OS/2 2.0 are shown in Figure 3-1 and Table 3-1 on page 3-2.

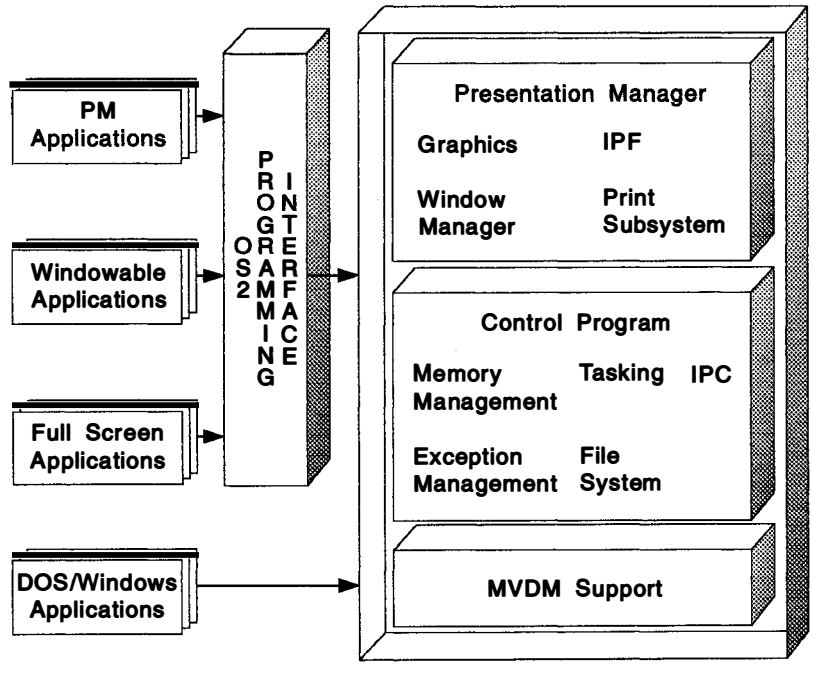



*Figure 3-1. OS/2 2.0 Application Types*

| *Table 3-1. OS/2 2.0 Application Types* | |
|---|---|
| **Full-screen** | Runs in non-PM session<br>Predominantly text-based<br>Direct hardware access<br>May use hardware text modes<br>May use restricted PM functions<br>DOS programming model |
| **Windowable** | Runs in a PM session<br>Text only<br>Hardware access using OS/2 APIs<br>May use hardware text modes<br>May use restricted PM functions<br>DOS programming model |
| **PM** | Runs in a PM session<br>May use advanced text capabilities<br>May use PM graphics functions<br>PM programming model |
| **DOS/Windows** | Runs in DOS (V86) session<br>Text and graphics<br>Direct hardware access<br>May use hardware text modes<br>DOS programming model |

## Full-Screen Applications

Full-screen applications do not run in the PM windows. A full-screen application is any OS/2 application that does not create a PM message queue. In other words, it is an application that does not rely on the PM mouse and keyboard processing for input.

Most full-screen applications use the Dos functions to perform input, output, memory management, and other activities. Full-screen applications also commonly use the standard-input, standard-output, and standard-error files created for them when they start. They can also use PM functions that do not require a message queue.

Full-screen VIO applications use the video (Vio), mouse (Mou), and keyboard (Kbd) functions to take complete control of the screen and input devices. Taking complete control is useful for programs that provide their own windowing system or their own high-speed graphics package. Although the video, mouse, and keyboard functions are not available as 32-bit functions under OS/2 2.0, full-screen VIO applications can still use some of the 32-bit features of OS/2 2.0.

## Windowable Applications

A windowable application is a full-screen application that also can run in a PM window. Although the application runs in a window, it does not create the window. Instead, the system creates the window and provides the input and output to the application just as if it were running in a full-screen session. A full-screen application can run in a window only if it does not use functions that directly access the devices that PM controls. For example, an application that attempts to retrieve the address of the video buffer or to change video modes cannot be made windowable.

Windowable applications do not take direct advantage of menus, icons, screen, mouse, or other features available in PM. They are usually command-line programs or simple text programs, and they are often ported from operating systems that have text-based interfaces, such as the C library input-and-output routine, printf. Windowable applications can use some of the 32-bit features of OS/2 2.0.

## PM Applications

A PM application is any OS/2 application that creates a message queue. Because a window is the only means a PM application has to receive input and display output, PM applications create one or more windows to interact with the user.

All OS/2 PM applications have essentially the following structure:

- A *main* procedure
- One or more *window* procedures
- Optional functions to support the main procedure and/or the window procedures

Figure 3-2 shows an PM application template.

```
Main Procedure
{
Initialize PM anchor block          (WinInitialize)

Create a message queue              (WinCreateMsgQueue)

Register the window procedure for
   application's window class       (WinRegisterClass)

Create the application's frame
   window, defining its controls,
   client window and resources      (WinCreateStdWindow)

Process messages from the
   application queue and direct them
   to the window procedures.        (WinGetMsg,
                                     WinDispatchMsg)
Close and destroy the application
   resources                        (WinDestroyMsgQueue,
                                     WinDestroyWindow,
                                     WinTerminate)
}

Window Procedure
{
Process messages or pass them on to
   the default window procedure.    (WinDefWindowProc)
}
```

*Figure 3-2. PM Application Template*

Because nearly all PM applications create and use windows, the main function carries out the same basic tasks in most applications. The typical main function does the following:

- Register each thread that calls PM functions. Threads are registered with the system by calling the WinInitialize function. This function creates an anchor block and returns an anchor-block handle that the thread can use in subsequent functions.

  **Note:** An anchor block is a data area reserved for internal PM resources for each thread that makes calls to PM functions. This data area is defined and managed by PM. It includes instance data in which to store the process's environment and storage for error messages.

- Create the message queue by using the WinCreateMsgQueue function. This function returns a queue handle that can be used in subsequent functions. When the queue is created, the application can register a window class, create a window, and start the message loop.
- Enter the main message loop. The application waits there for messages to appear in the queue, retrieves them, and dispatches them, as appropriate, to its windows. When the user or system chooses to terminate an application, a WM_QUIT message is used to trigger an exit from the message loop.
- Carry out various termination activities, including destroying windows, releasing memory, destroying message queues, closing files, and severing connections with the shell and other applications.

Applications written for PM have full access to the complete set of user interface tools: menus, icons, scroll bars, and so on, and often present a WYSIWYG (what-you-see-is-what-you-get) view of their data. PM applications often make extensive use of a mouse and display and have access to all the 32-bit features of OS/2 2.0.

## DOS/Windows Applications

Most DOS/Windows** applications can be run in an OS/2 2.0 DOS session, in the virtual 8086 mode of the 80386 microprocessor. The virtual 8086 mode provides a high degree of I/O and memory protection, so that the crash of a DOS/Windows application running in an OS/2 2.0 DOS Session does not crash the entire operating system as well. DOS/Windows applications can be run in full-screen or windowed DOS Sessions. Multiple DOS/Windows applications can run concurrently, with background DOS/Windows applications not suspended, but multitasked like OS/2 applications. DOS/Windows applications can also communicate with PM through the system clipboard, and to other applications through named pipes.

## Programming Models

OS/2 2.0 applications can be further classified according to the manner in which they are built, as shown in Table 3-2.

*Table 3-2. OS/2 Version 2.0 Programming Models*

| | |
|---|---|
| **Pure 16-Bit** | Written in a 16-bit language<br>Built using 16-bit tools<br>16-bit EXE format |
| **Mixed 16-Bit** | Written in 16-bit C<br>Built using 16-bit and 32-bit tools<br>32-bit EXE format |
| **Pure 32-Bit** | Written in 32-bit C<br>Built using 32-bit tools<br>32-bit EXE format |
| **Mixed 32-Bit** | Written in 16-bit and 32-bit C<br>Built using 16-bit and 32-bit tools<br>32-bit EXE format |

A pure 16-bit OS/2 application can be run on a 16-bit or 32-bit OS/2 system. Mixed 16-bit, mixed 32-bit, and pure-32-bit applications can be run only on a 32-bit OS/2 system.

## Pure 16-Bit Applications

As shown in Figure 3-3 on page 3-6, pure 16-bit OS/2 applications:

- are compiled with a 16-bit C compiler
- use 16-bit C runtime libraries
- may be a small/medium/large/huge model
- use 16-bit OS/2 API
- use 16-bit OS2.H include file
- are linked with 16-bit version of OS2.LIB called OS2286.LIB
- are linked with 16-bit linker
- have the 16-bit EXE format

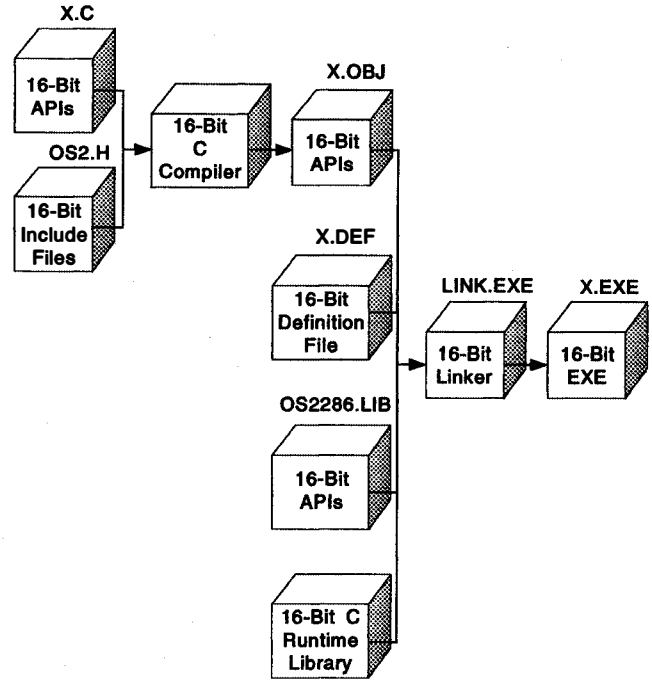



*Figure 3-3. Building a Pure 16-Bit Application*

These applications can be full-screen, windowable, or PM applications. They can run under the 16-bit version of the operating system, as well as under the 32-bit version of the operating system (that is, on machines with 80286 or 80386 microprocessors). However, they cannot take advantage of the features of the 32-bit programming environment. They use the 16-bit segmented memory model, and, therefore, do not have access to the entire 32-bit virtual address space.

## Mixed 16-Bit Applications

As shown in Figure 3-4 on page 3-7, mixed 16-bit OS/2 applications:

- are compiled with a 16-bit C compiler
- use 16-bit C runtime libraries
- may be a small/medium/large/huge model
- use 16-bit OS/2 API
- may use 32-bit OS/2 API
- use 16-bit OS2.H include file
- are linked with 16-bit version of OS2.LIB called OS2286.LIB
- are linked with 32-bit linker
- have the 32-bit EXE format

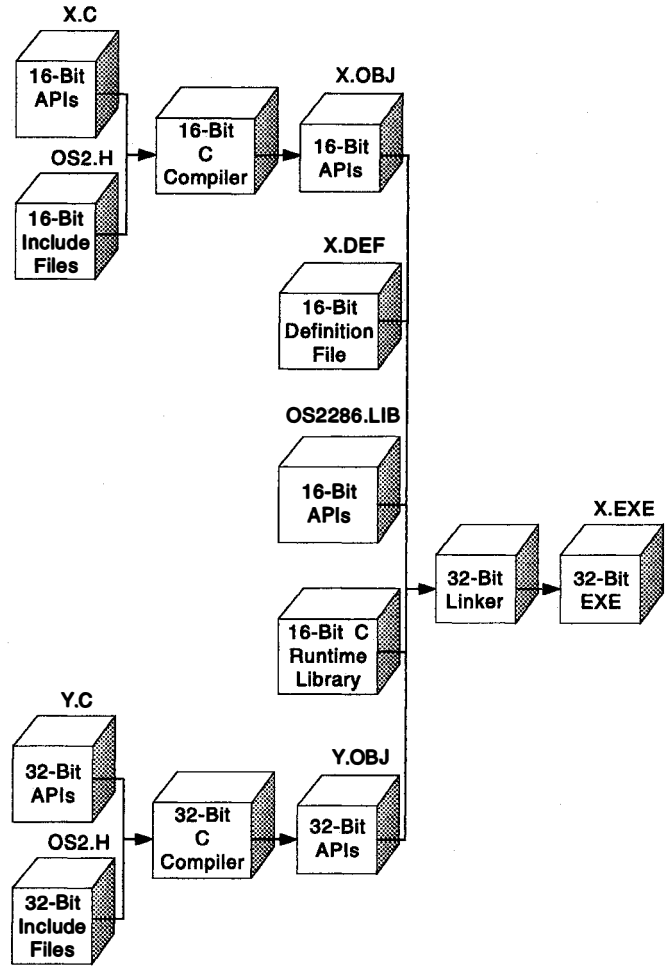



*Figure 3-4. Building a Mixed 16-Bit Application*

These applications can be full-screen, windowable, or PM applications. Like pure 16-bit OS/2 applications, they do not have access to the entire 32-bit virtual address space. Because they have a 32-bit EXE format, they must run under the 32-bit version of the operating system. This enables them to run faster and more efficiently.

## Pure 32-Bit Applications

As shown in Figure 3-5 on page 3-8, pure 32-bit OS/2 applications:

- are compiled with a 32-bit C compiler
- use 32-bit C runtime libraries
- are small model only
- use 32-bit OS/2 API
- use 32-bit OS2.H include file
- are linked with 32-bit OS2386.LIB
- are linked with 32-bit linker
- have the 32-bit EXE format

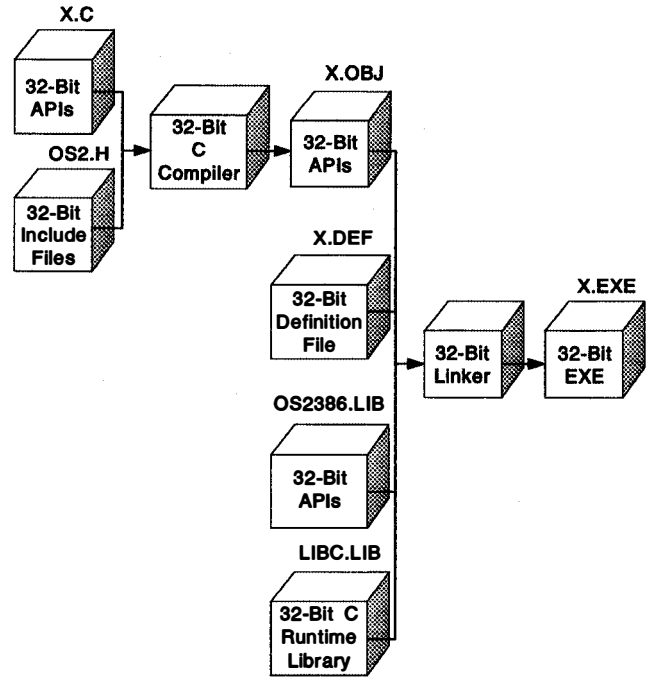



*Figure 3-5. Building a Pure 32-Bit Application*

These applications can be windowable, or PM applications. Because they have a 32-bit EXE format, they can run only under the 32-bit version of the operating system. Pure 32-bit applications incorporate the 32-bit, flat memory model and memory protection mechanisms which are common on a wide range of computer industry hardware platforms. This feature allows pure 32-bit applications to be more easily ported to other platforms.

## Mixed 32-Bit Applications

As shown in Figure 3-6 on page 3-9, mixed 32-bit OS/2 applications:

- are compiled with a 32-bit C compiler with the variables aligned in such a way that they can be passed to 16-bit functions (with IBM C Set/2*, for example, you would use the /Gt + flag on the command line).
- use 32-bit C runtime libraries
- are small model only
- use 32-bit OS/2 API
- may use 16-bit OS/2 API
- may use only 16-bit API or 32-bit API in a single .C file, but not both because of include file support
- use 32-bit OS2.H include file
- are linked with 32-bit version of OS2386.LIB
- are linked with 32-bit linker
- have the 32-bit EXE format

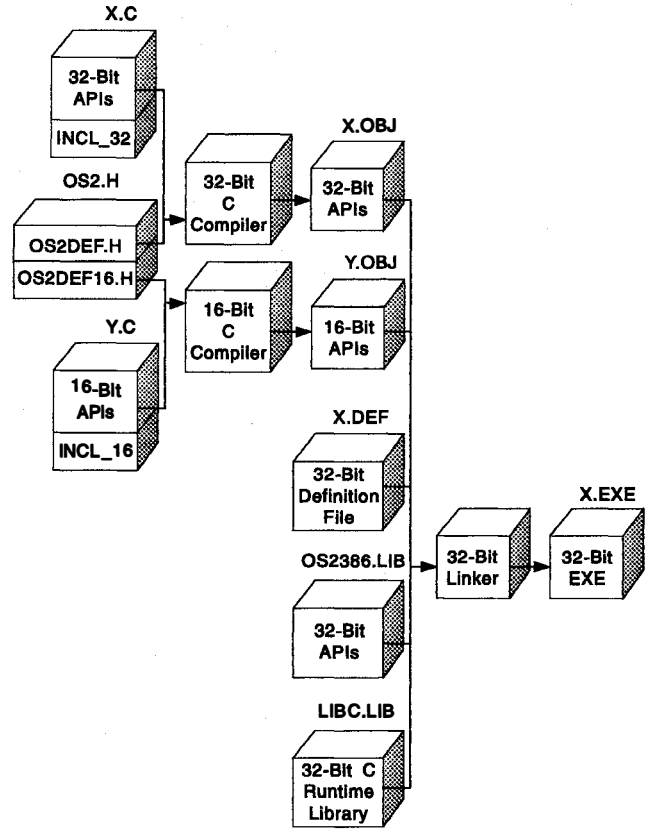



*Figure 3-6. Building a Mixed 32-Bit Application*

These applications can be full-screen, windowable, or PM applications. Because they have a 32-bit EXE format, they can run only under the 32-bit version of the operating system. Mixed 32-bit OS/2 applications can access the entire 32-bit virtual address space.

## The Program Development Environment

OS/2 2.0 provides different names and entry points for all 16 and 32-bit APIs. Therefore, it is possible to mix 16- and 32-bit code within a single .EXE module. It is also possible to call 32-bit APIs from a 16-bit C program (mixed 16-bit programming model), and to call 16-bit APIs from a 32-bit program (mixed 32-bit programming model). In order to support this, two different libraries are provided with OS/2 2.0.

**OS2286.LIB** This library links to 16-bit pure/mixed applications. It contains links to the 16-bit ordinals using the 16-bit DosXXX names, and links to the 32-bit API entry points using the 32-bit Dos32XXX names.

**OS2386.LIB** This library links to 32-bit pure/mixed applications. It contains links to the 32-bit ordinals using the the DosXXX or Dos32XXX names, and links to the 16-bit APIs using the Dos16XXX names.

Linking with these libraries allows the 16- and 32-bit versions of a given API to be called from the same .EXE module. The names used in the application must match these names by link time. However, language pre-processors can be used to hide some of this from the application programmer.

When you compile a program, use the appropriate compiler option to ensure that all variables defined in that compilation unit are guaranteed not to cross 64K boundaries. The option that directs your compiler to use 16-bit versions of the malloc family functions (calloc, malloc, realloc, and free) should be used as well.

**Note:** When you use the 16-bit enabling option, data items larger than 64K in size will be aligned on 64K boundaries, but will also cross 64K boundaries.

## Include File Architecture

Only 16-bit APIs or only 32-bit APIs can be used in a single C module. To understand why, let's look at the OS/2 include file layout. The OS2.H file includes OS2DEF.H before including the appropriate BASE and PM header files. OS2DEF.H contains typedefs and macros for most of the system related objects, structures, and data types used when accessing the APIs. Programmers should use the same type names from the include files in both the 32-bit and 16-bit environment. For example, when the code contains:

```
PSZ     buffer = "hello";
```

in the 16-bit system, OS2DEF.H changes it to:

```
char far   *buffer = "hello";
```

However in the 32-bit system the declaration will be converted to:

```
char    *buffer = "hello";
```

because of the flat memory model.

OS/2 2.0 provides support for the following 16-bit subsystems:

- Vio (video functions)
- Kbd (keyboard functions)
- Mou (mouse functions)
- Mon (monitor functions)

If you wish to use any other 16-bit API functions in your 32-bit code, you must provide the appropriate header file(s). The template for the OS2.H include file is shown in Figure 3-7.

```
#define OS2_INCLUDED /* Indicating that this file has been included */

/* Common definitions */
#include <os2def.h>

/* OS/2 Base Include File */
#ifndef INCL_NOBASEAPI
   #include <bse.h>
#endif /* INCL_NOBASEAPI */

/* OS/2 Presentation Manager Include File */
#ifndef INCL_NOPMAPI
   #include <pm.h>
#endif /* INCL_NOPMAPI */
```

*Figure 3-7. Template for the OS2.H Include File*

## C Compiler Support

The IBM C Set/2 compiler provides the capability of calling 16-bit API functions and using 16-bit far pointers (16:16) from flat model 32-bit applications. This is done by providing three new keywords: _Seg16, _Far16, and _Pascal. The _Seg16 keyword is used to declare 16:16 pointers, while the _Far16 and _Pascal keyword are used to declare 16-bit far function references and to generate the 16-bit far calling sequence in 32-bit code.

The samples that follow were written using IBM C Set/2 keywords. If you use a different compiler, use the keywords your compiler employs to achieve the same effect.

To declare a pointer as 16:16, you would use the _Seg16 keyword:

```
char * _Seg16 foo;
```

(Notice that _Seg16 comes after the *, not before.) This declares *foo* to be a segmented pointer. In any operations involving the pointer, the pointer is converted to a flat pointer prior to the operation and converted back into a segmented pointer (if necessary) after the operation is complete. For example, for something like

```
foo[0]='x';
```

the pointer *foo* is first converted to a flat pointer, then "x" is placed into the first element, rather than using the segmented pointer directly. This means that a 16-bit routine can use the variable *foo* directly, since the value in the pointer is stored in segmented form already.

Flat pointers also can use segmented pointers. When a flat pointer is passed to a 16-bit routine, the pointer is automatically converted to 16:16 as it is passed.

Note that pointers that are declared as _Seg16 are stored as 16:16, but dereferenced as 0:32. Since all 16-bit objects are pseudo-tiled, this increases protection while sacrificing speed.

Output parameters are converted. If a 32-bit application calls a 16-bit API, and the API returns a 16:16 pointer, the output pointer is converted to the 0:32 format.

Regardless of parameter conversions, the C compiler generates the 16-bit calling sequence when encountering a "_Far16 _Pascal" function call type. The compiler calls a library function to switch to a 16-bit stack, push parameters and do parameter conversions, execute the 16-bit call, switch back to the 32-bit stack, and return to the 32-bit application. Figure 3-8 on page 3-12 illustrates the stack frame generated for the VioXXX example.

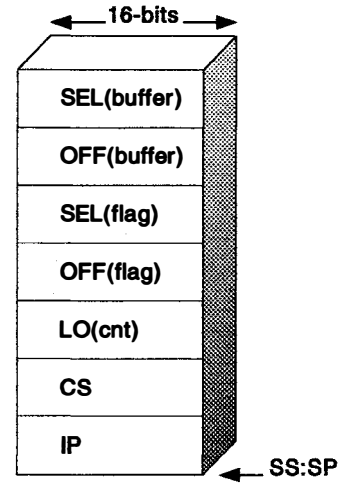



*Figure 3-8. Stack frame for VioXXX Example*

## Library Support

C Set/2 has two sets of .LIB files: a 16-bit and a 32-bit version. The major difference between the two libraries is how the default name of an API function is handled, that is, what an API function reference is mapped to.

# Mixing 16-Bit and 32-Bit Code

OS/2 2.0 supports 32-bit programs, 16-bit programs, and mixed programs (programs that contain both 16-bit and 32-bit code). This section deals with mixed program development.

## Thunking

All 32-bit OS/2 code can call 16-bit OS/2 code by conforming to the existing conventions in 16-bit OS/2's segmented dynamic link model. The only 16-bit modules a 32-bit application should use are the VIO, KBD, MOU, or other 16-bit modules for which suppliers have not provided 32-bit support. 32-bit applications that use 16-bit functions may not run as fast as they would if they used 32-bit functions. 16-bit OS/2 code can also utilize 32-bit code by conforming to the 32-bit conventions. Code that is used to "glue" 16 and 32-bit routines is commonly known as a *thunk*. A thunk can reside in an application or in a library module. A 32-bit thunk binds 32-bit code to 16-bit code. A 16-bit thunk binds 16-bit code to 32-bit code. Figure 3-9 on page 3-13 illustrates how 16- and 32-bit applications and subsystems interact through the use of thunks.

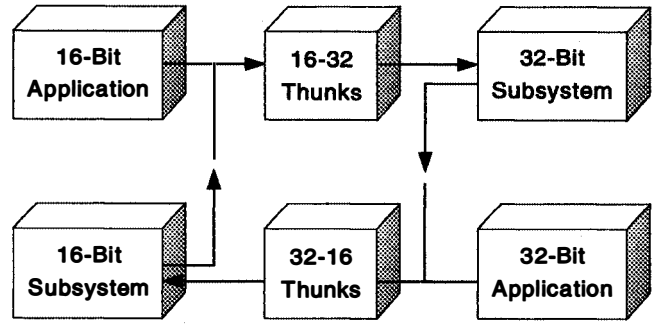



*Figure 3-9. 16 to 32-Bit Application/Subsystem Interactions*

In OS/2 2.0, the operating system supplies an interface between 16-bit and 32-bit code, called a *thunking layer*. The purpose of the thunking layer is to convert code and memory objects from 16-bit to 32-bit and back. This conversion is transparent to the callee and caller.

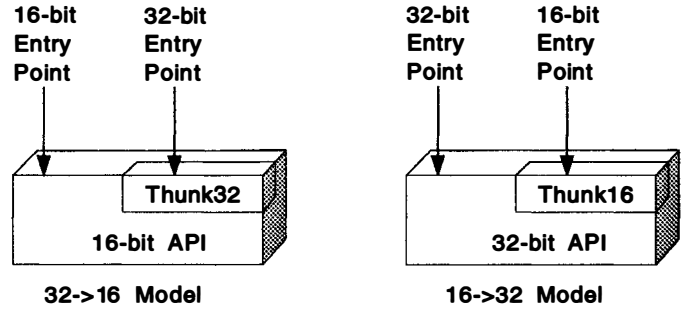



*Figure 3-10. Thunk Models*

Most of the system-supplied thunks are of the 32→16 variety. A 32→16 thunk must perform the following functions when calling 16-bit code from 32-bit code.

1. Save the FLAT context

   The thunk should preserve the current FLAT context, which involves saving the current DS and SS selectors. The values in EBX, ESI, and EDI also need to be preserved because the 16-bit PM conventions do not preserve them. Also, the 32-bit calling convention saves EBX, ESI, and EDI. The 16-bit calling convention only saves EDI and ESI, so EBX would be lost.

   You should also save EBP, since the 16-bit function only uses BP.

2. Copy the parameters

   The parameters (or call frame) need to be reconstructed. Parameters may need to be aligned on the stack. The flat memory environment usually pushes 32-bit, DWORD sized items. The segmented memory environment uses 16-bit, WORD sized items.

   Copying parameters consists of:

   - Checking for boundary crossings
   - Converting pointers
   - Pushing the parameters in the proper order

3. Convert the stack to 16:16

   The 16-bit function expects the stack to be addressable by SS:SP and BP. The flat memory environment is passing in a ESP, EBP addressable stack. The stack pointer, SS:SP is created by calling the API function, DosFlatToSel on the current ESP. (See "DosFlatToSel" on page 3-17 for more information on DosFlatToSel.)

4. Call the 16-bit routine

   The thunk then executes a far 16:16 call to the target routine.

5. Restore the flat context

   The far 16:16 call will return to the thunk when the call is complete. The thunk needs to restore all segment registers: SS is retrieved, the DS value is restored into both DS and ES, and ESP and EBP are both restored. The saved registers EDI, ESI and EBX are also restored.

   **Note:** It is necessary to convert any parameters upon returning.

6. Return to caller

A 16→32 thunk must perform the following functions when calling 32-bit code from 16-bit code.

1. Save current context

   All the normal segment registers are about to be destroyed, so they should be saved; SS and DS should be saved, and ES if needed. The extended registers will be saved by the callee.

2. Copy the parameters

   Align parameters into DWORD sized parameters, if necessary.

3. Setup a flat context

   The 32-bit routine expects SS:ESP as the stack selector. Use the DosSelToFlat API function on SS:SP to create an ESP. (See "DosSelToFlat" on page 3-17 for more information on DosSelToFlat.) Then load the flat selectors into SS, DS, and ES.

4. Call the 32-bit routine

   Perform a near (0:32) call to the 32-bit function.

5. Restore saved context

   Restore SS:SP and DS.

6. Return to caller

One of the major goals of OS/2 2.0 is to provide compatibility between 16-bit and 32-bit versions. Thunks provide this compatibility, by dealing with the following issues:

- OS/2 Memory Layout

  OS/2 2.0 uses a flat linear (0:32) addressing model. The 16-bit OS/2 code uses a selector:offset (16:16) addressing model. The thunk layer must convert the addresses between the two models.

- Different Parameter Sizes

  The 32-bit version of the operating system uses 32-bit (LONG or DWORD) values as the basic data type. The 16-bit version uses a 16-bit (SHORT or WORD) value as the basic data type. The thunk layer must convert between WORD and DWORD length data.

The 32-bit stack is DWORD-based. The 16-bit stack is WORD-based. The thunk layer must make a new copy of the parameters on the stack and realign when needed.

- 64K Segment Boundary Problem

  16-bit code can only address up to 64K in any one segment. The only limit on 32-bit code is the maximum size of the linear address space (4GB). This creates a problem when a 32-bit data item is larger than 64K and is being passed to 16-bit code. The thunk layer must make the data item addressable by the 16-bit routine.

- Different Call Models

  The 0:32 addressing model uses a near call for all functions. The 16:16 model uses a far call for all functions. A problem arises when a thread of one model tries to call a procedure of the other model. The two models push different return addresses on the stack. The thunk layer must produce the correct calling sequence.

## 32-Bit OS/2 Memory Layout

This section briefly describes the memory layout for OS/2 2.0, and how the segmented memory environment is mapped to the flat memory environment.

### Flat Memory

The 32-bit addressing of OS/2 2.0 is accomplished on the 386 by creating a very large segment (up to 4GB in size), and using all near addresses inside this large segment. A data and a code segment are mapped to this large address space, with their limits set to 4GB (0xfff...), as shown in Figure 3-11. The selectors for this are allocated in the global descriptor table (GDT), and are called the "flat" selectors.

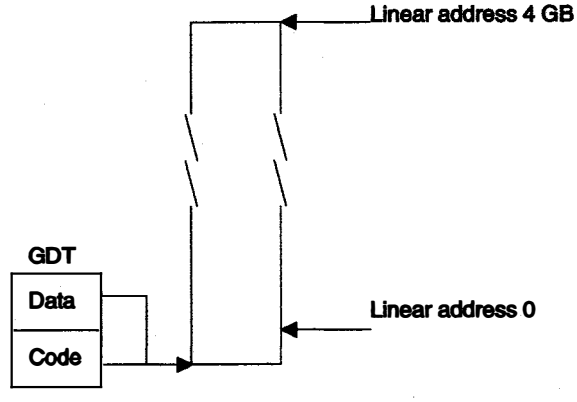



*Figure 3-11. Mapping the flat Address Space Using GDT Selectors*

All addresses in the flat memory model are 32-bit offsets into these two segments. The base addresses of the two selectors are always zero, and therefore the flat offsets are the same as the linear addresses.

### Tiled Memory

In order for OS/2 2.0 to be compatible with 16-bit applications, the 16-bit environment can be mapped to the 32-bit environment in such a way that addresses can be quickly converted between the two environments.

This mapping is accomplished by a technique called *tiling*. A tiled local descriptor table (LDT) contains up to 8192 descriptors, where the segment base address in each descriptor is a multiple of 64KB. In this manner, each descriptor points to a 64KB region of memory. Contiguous descriptors map into a contiguous linear address space. Because the LDT only holds 8192 selectors, the maximum address allowed in the tiled memory region is 512MB. Figure 3-12 shows the way in which memory addresses within the tiled LTD are mapped into the process address space. This tiled region of memory is called the *compatibility region*.

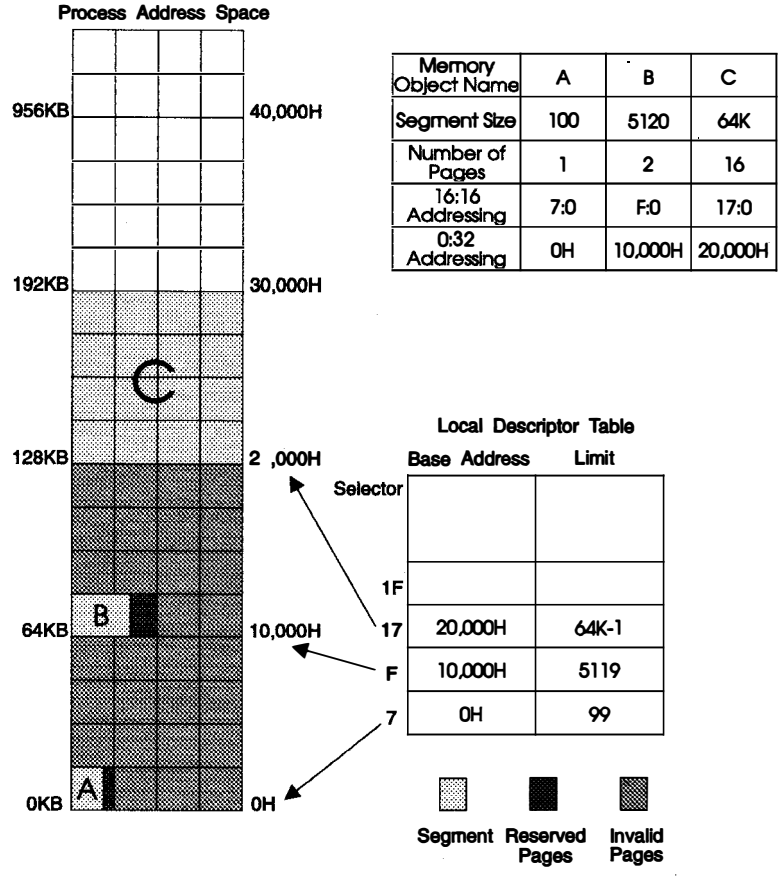



*Figure 3-12. 16-Bit Address Space Mapped to the Flat Address Space*

The addresses within the process address space can be referenced by applications or modules using 16:16 addressing, in a manner similar to previous versions of the operating system. However, the same physical memory locations can also be accessed by 32-bit applications and modules using the 0:32 addressing method. Both the LDT entries used for 16:16 memory addressing and the page table entries used for 0:32 memory addressing can translate to the same memory locations. This enables 32-bit applications to make use of 16-bit modules and resources, and allows 32-bit and 16-bit applications to coexist and communicate with one another.

LDTs are managed in a different way from previous versions of the operating system. Each LDT is allocated a sparse memory object until descriptors are inserted upon loading an application. Descriptors for shared memory objects are inserted downwards starting at the top of the LDT, whereas private memory descriptors are inserted upwards starting at the bottom of the LDT. This reflects the

management of the linear address space by the operating system. Therefore, the minimum LDT size is 8KB, using one page for the shared descriptors and one page for the private descriptors. Note that each code or data selector reserves a full 64KB of linear address space.

This is equivalent to the implementation used in the previous versions of the operating system. It must be noted, however, that a memory object greater than 64KB require special handling when used.

The following memory objects use LDT descriptors:

- 16-bit .EXE files
- 16-bit .DLL files
- DosAllocSeg() calls
- DosAllocMem() calls with tiling
- 32-bit .EXE files with tiling
- 32-bit .DLL files with tiling

Important notes about tiled memory:

- Only LDT selectors are tiled. This means that only selectors allocated from the LDT can be converted.

A memory object allocated in the compatibility region has both a 16:16 address and a 0:32 address, allowing access by applications using either addressing method. All 32-bit executable modules can therefore create 16:16 aliases for memory objects in the compatibility region and conversely, 16-bit modules can create 0:32 bit aliases. The two types of addresses are related by the following API functions:

- DosSelToFlat
- DosFlatToSel

***DosSelToFlat:*** The DosSelToFlat API function converts a 16:16 address to a 0:32 address. This function is described as follows:

```
PVOID DosSelToFlat (ULONG ulSelector);
```

*Figure 3-13. Using DosSelToFlat*

This function takes a 32-bit selector value (*ulSelector*) and returns a pointer to the 32-bit flat address.

***DosFlatToSel:*** The DosFlatToSel API function converts a 0:32 address to a 16:16 address. This function is described as follows:

```
ULONG DosFlatToSel (PVOID pFlatAddress);
```

*Figure 3-14. Using DosFlatToSel*

This function takes a pointer to a 32-bit flat address (*pFlatAddress*) and returns a 32-bit selector value.

## Different Parameter Sizes

The most significant problem with different parameter sizes is the packing of structures. If there are two different structures for 16 and 32 bit with the same name and it is used in the call, then the thunk has to take this into account when passing the information. All the problems in this area can be eliminated if any structure that you define in 32-bit code is made up of only DWORDS.

## 64K Segment Boundary Problems

The mixed model programmer must be aware of potential problems when converting addresses between the two addressing models.

Consider the following scenario:

1. A 32-bit program passing a pointer to a 16-bit routine.
2. The object referenced by this pointer crosses a 64K tile boundary.
3. The pointer is converted using DosFlatToSel.

When the 16-bit routine tries to reference the data object, it will be unable to access the part of the object past the 64K tile boundary, as illustrated in Figure 3-15.

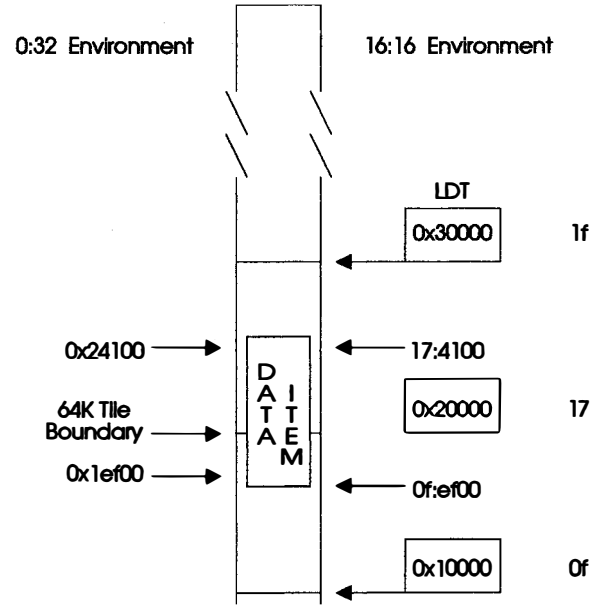



*Figure 3-15. Data Item Spans 64K Tile Boundary*

In Figure 3-15, the data object starts at 32-bit address 0x1ef00, and ends at 0x24100. When the starting address is converted, it produces the 16:16 address 0f:ef00. However, the length of the data item is 0x5200. The 16:16 routine will be unable to address the end of the object, because its offset is greater than 64K, which is larger than the 16-bit segment limit.

To make this data item addressable, the caller will have to copy it to another location in memory that will not cross a boundary.

## Different Call Models

In a 16-bit far call, CS and IP are pushed when the call is made. In a 32-bit segment call, CS and EIP are pushed. To assure that the correct return address is on the stack, the 32→16 thunk jumps to a 16-bit segment before making a 16:16 call. When the call returns, the thunk makes a far jump to 32-bit code that continues the processing. If the call had been made from the 0:32 segment, the stack would not have had a valid return address on it, since EIP would have been pushed. For 16:16→0:32 thunks, in order to get the correct return address, the thunk jumps to 32-bit code to make the call.

Another consideration in the call model is the calling convention. A 16-bit application might very well be using the Pascal calling convention with the 32-bit code using the C calling convention. It is obvious that the parameters need to be pushed on the stack to conform to the convention that is used.

## Calling 16-Bit Code from 32-Bit Code

This section describes some of the conventions used for calling 16-bit code from 32-bit code.

The samples that follow were written using IBM C Set/2 keywords. If you use a different compiler, use the keywords your compiler employs to achieve the same effect.

IBM C Set/2 provides three linkage conventions for calling 16-bit code. These are:

- far16 cdecl
- far16 fastcall
- far16 pascal

For example, a function Func can be declared as a 16-bit function using the far16 pascal linking convention.

```
long _Far16 _Pascal Func (short, char *);
```

*Figure 3-16. Using _Far16 _Pascal to Declare a 16-Bit Function*

The _Seg16 type qualifier is used to declare 16-bit pointers. For example,

```
char * _Seg16 p16;
```

*Figure 3-17. Using _Seg16 to Declare 16-Bit Pointers*

declares p16 as a segmented pointer that can be used directly by a 16-bit application.

The #pragma seg16 directive is used to ensure that shared data items do not cross 64K boundaries. The #pragma seg16 directive can be used directly with the data item or through a typedef. The following code fragment shows both ways of using this directive:

```
struct family {
   long          john;
   double        carolynn;
   char * _Seg16 geoff;
   long          colleen;
};

#pragma seg16 (cat)
struct family cat;                  /* cat is qualified directly */

typedef struct family tom;
#pragma seg16 (tom)
tom edna;                           /* edna is qualified using a typedef */
```

*Figure 3-18. Using the #pragma seg16 Directive*

## Using the _Seg16 and _Far16 _Pascal keywords

The keywords, _Seg16 and _Far16 _Pascal, are used to call 16-bit routines from 32-bit modules. These keywords are:

- only available in modules compiled as 32-bit
- used to convert pointers into their selector:offset representation
- used to modify function prototypes

```
char * _Seg16 Ptr;
char *Ptr32;
int main(void)
{
        Ptr = "A B C D";
        Ptr32 = "A B C D";
        return 0;
}
```

*Figure 3-19. Using _Seg16*

In Figure 3-19, Ptr gets a 16:16 address to the string, while Ptr32 gets a 32-bit flat address to the string. When a 32-bit module dereferences a _Seg16 pointer, an inline conversion is generated before the memory is accessed.

When _Far16 _Pascal is used on a function prototype, IBM C Set/2 will produce a very simple thunk to get you to the 16-bit routine.

```
extern short _Far16 _Pascal m16();
```

Figure 3-20 on page 3-21 illustrates that _Far16 _Pascal is only valid inside 32-bit modules.

```
char * _Seg16 pchPtr;

extern short _Far16 _Pascal DosWrite(short,
                                     char *,
                                     short,
                                     short *);

int main(void)
{       short cbWrite;

        pchPtr = "Hello World\n";

        /* If using the DosWrite function, you must link with  */
        /* OS2286.LIB, or else use the Dos16Write function     */

        DosWrite(0,pchPtr,strlen(pchPtr),&cbWrite);

        return 0;
}
```

*Figure 3-20. Using _Far16 in a 32-Bit Module*

## Formal Parameters

When calling a _Far16 _Pascal function, the IBM C Set/2 compiler will use WORD size parameters, instead of the default DWORD size (if the function prototype calls for WORD parameters).

```
extern long DosBeep(long,long);
extern short _Far16 _Pascal Dos16Beep(short,short);
int main(void)
{
        Dos16Beep(400,50);      // 16-bit DosBeep
        DosBeep(300,100);
        return 0;
}
```

*Figure 3-21. Passing Different Size Parameters*

In Figure 3-21 a 16-bit 400 and a 16-bit 50 will be pushed on the stack. In contrast, the 32-bit call to to DosBeep will push a 32-bit 300 and 100.

As mentioned previously there are two major problems to handle when passing data to a 16-bit module: objects crossing 64K segment boundaries and the WORD vs DWORD alignment in structures.

***The 64K Segment Boundary Problem*** Using the IBM C Set/2 compiler, you can alleviate the 64K segment boundary problem when the data object is smaller than 64KB by using the /Gt+ switch to control alignment.

- No statically allocated data will cross a 64K boundary when the /Gt+ switch is specified. This means the compiler will never declare global or static variables that cross 64K boundaries.
- Local variables in a function that calls a 16-bit function will not cross a 64K boundary. Any memory that you allocate will not cross a boundary when the /Gt+ switch is specified.

- No system allocated memory will cross a boundary. Every data object that you allocate using DosAllocMem will always start on a 64K boundary. If the item is smaller than 64K, then the item will never cross a boundary, but only if the OBJ_TILE flag is given.

However, if you allocate a very large buffer, and pass a pointer into the buffer that is near a 64K boundary, the 16-bit routine will be unable to fully address the object.

Function prototypes should be declared as follows:

```
extern short _Far16 _Pascal m16(short, char *);
```

noting the following:

- Any pointer parameters do not have to be explicitly declared as _Far16 _Pascal. The compiler will assume that pointers are supposed to be _Far16 _Pascal.
- If you pass a _Seg16 pointer to a 32-bit routine, you should be sure that a function prototype for the 32-bit routine exists. Also, make sure that the parameter is declared as a pointer. If there is no prototype, then you should explicitly cast the pointer to be a normal 32-bit pointer. For example: `printf("%s,"(char *) pFar16Ptr)`

***Structure Alignment:*** There are structures in the system, and especially in PM, that have different packing between their 16- and 32-bit representations.

DWORD alignment in C language means items that are DWORD in length, such as pointers, will always be aligned on 4 byte boundaries. WORD alignment means that items that are WORD size or greater will be aligned on 2 byte boundaries. DWORD alignment does not mean that all structure members are DWORD aligned. Refer Figure 3-22 for an example.

```
                        Offsets for

                        WORD    DWORD
typedef struct _K {
        char c;         0       0
        char c2;        1       1
        long b;         2       4
        short a;        6       8
        short d;        8       10
        } K;
```

*Figure 3-22. Structure Alignment*

Notice that *c* and *c2* are byte-sized items, and therefore do not qualify for alignment. Notice that *b* is aligned on a WORD boundary for WORD alignment, and DWORD boundary for DWORD alignment. Elements *a* and *d* are WORD aligned in both alignment types.

Table 3-3 categorizes alignment by data size.

*Table 3-3. WORD/DWORD Alignment*

| Alignment | BYTE | WORD Data Size | DWORD Data Size |
|---|---|---|---|
| WORD | BYTE | WORD | WORD |
| DWORD | BYTE | WORD | DWORD |

Problems with alignment result when data is shared between 32-bit and 16-bit modules, or when calling 16-bit subsystems, such as VIO or KBD. For example, consider the code in Figure 3-23.

```
typedef struct _KBDKEYINFO {    // Offsets
    UCHAR  chChar;              // 0
    UCHAR  chScan;              // 1
    UCHAR  fbStatus;            // 2
    UCHAR  bNlsShift;           // 3
    USHORT fsState;             // 4
    ULONG  time;                // 6    (32-bit offset is 8)
} KBDKEYINFO;

extern USHORT _Far16 _Pascal KBD16PEEK(KBDKEYINFO _Far16 *,USHORT);

main()
{
        KBDKEYINFO kbciKeyInfo;

        KBD16PEEK(&kbciKeyInfo, 0);
}
```

*Figure 3-23. Example of a Packing Problem*

In Figure 3-23, the last element of the KBDKEYINFO structure (*time*) has a different offset between the 32-bit program and the 16-bit API.

A number of things can be done to solve this problem:

- Change the packing of one of the modules using the IBM C Set/2 switch, /Sp.

  The compiler switch /Sp resets the default packing for the C compiler. The default packing for 16-bit programs is WORD alignment, and the default packing for 32-bit programs is DWORD alignment. Using this switch may cause problems if the module is going to call other systems. For example, using /Sp2 will fix the call to KBD, but will cause problems with calls to true 32-bit subsystems.

- Change the structure definition so that alignment does not affect the offsets of any elements. The easiest way to do this is to sort the elements by size so that DWORDS are first, followed by WORDS and then BYTES.

  This is the suggested solution if you have control over the data structures definition. However, here the structure is defined by the system.

- Use a #pragma to change the packing of a particular item, as in Figure 3-24 on page 3-24.

  In this situation, using the #pragma is the best choice. It has no impact on other data structures, and can be used to target specific structures.

```
#pragma pack(2)
typedef struct _KBDKEYINFO {    // Offsets
    UCHAR  chChar;              // 0
    UCHAR  chScan;              // 1
    UCHAR  fbStatus;            // 2
    UCHAR  bNlsShift;           // 3
    USHORT fsState;             // 4
    ULONG  time;                // 6    (32-bit offset is 8)
} KBDKEYINFO;

#pragma pack()    // Resets packing to the default value

extern USHORT _Far16 _Pascal KBD16PEEK(KBDKEYINFO _Far16 *,USHORT);
 ...
```

*Figure 3-24. Another Packing Problem*

Other alignment problems may develop in applications as well. Consider the following code fragment.

```
typedef struct _LineBuf {
        USHORT Row;
        USHORT Col;
        char *Text;
        USHORT Attr;
        } LineBuf;

LineBuf MainBuf[12];
```

*Figure 3-25. Encountering Alignment Problems*

Notice that the alignment of LineBuf is the same between 16-bit and 32-bit modules. However, the alignment is different when the structure element is in an array, as illustrated in Figure 3-26.

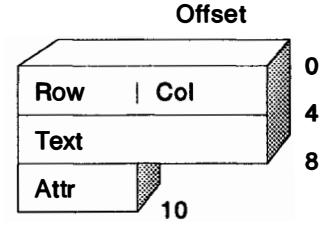



*Figure 3-26. An Array Structure Element*

For the element Text to always be DWORD aligned, each instance of the structure must begin on a DWORD boundary. When you have an array of these structures, the 32-bit compiler must pad out the array so that the alignment works out correctly, as illustrated in Figure 3-27 on page 3-25.

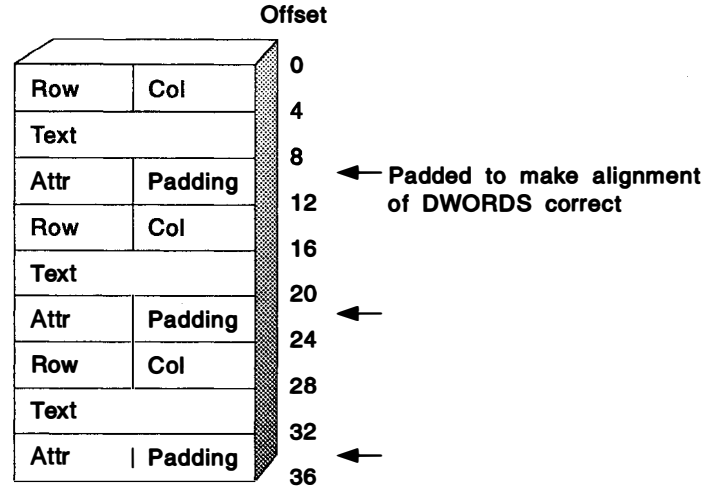



*Figure 3-27. Aligning an Array of Structures*

However, a 16-bit compiler does not do this for DWORD values. Hence the difference in the arrays. In general, if the structure has DWORD length items and the length of the structure is not a multiple of 4, then the structure will be padded if used in an array. The packing will need to be changed for addressing to work correctly. With the IBM C Set/2 compiler, use the #pragma pack(2) statement.

## Examples of using _Far16 _Pascal and _Seg16

Here are several example programs that make use of the _Far16 _Pascal and _Seg16 keywords.

```
/*
 *  32-16 example 1.  Compile using the /Ss option.
 */

#include <os2.h>
#include <stdio.h>

extern USHORT _Far16 _Pascal DOS16READ(unsigned USHORT,char *,
                               unsigned USHORT,unsigned USHORT *);

extern USHORT _Far16 _Pascal DOS16WRITE(unsigned USHORT,char *,
                               unsigned USHORT,unsigned USHORT *);

char *String="Enter Your Name: ";
char * _Seg16 String16;
```

```
int main(void)
{
   USHORT ret16,cbRead16;
   ULONG ret,cbRead;
   char buf[128];

   String16 = String;

   ret = DosWrite(0,String,strlen(String),&cbRead); // 32-bit call

   ret16 = DOS16READ(0,buf,127,&cbRead16);   // 16-bit call

   ret16 = DOS16WRITE(0,buf,cbRead16,&cbRead16); // 16-bit call

   return 0;
}
```

*Figure 3-28 (Part 2 of 2). Using the _Far16 _Pascal and _Seg16 Keywords*

This example program has several features that should be noted.

- Function prototypes

  At the start of the file, the standard <OS2.H> file is included. This will give the function prototypes, typedefs, and structure definitions for the standard 32-bit APIs. You must create your own 16-bit function prototypes for all 16-bit functions you plan to use.

- Names of 16-bit routines

  Note the use of DOS16 as the prefix for the 16-bit calls. The library files for OS/2 2.0 have been expanded to include these new names. They are intended to remove conflicts between names. In the example program above, there are calls to both the 16- and 32-bit versions of DosWrite. The only way to differentiate between the 16- and 32-bit versions of the API functions is by appending the 16 to the component name.

The following example has a slightly more complicated structure.

```
/*
 * 32-16 example 2.  Compile using the /Ss option.
 */
#include <os2.h>
#include <stdio.h>

#pragma pack(2)

typedef struct _KBDKEYINFO {     // Offsets
    UCHAR  chChar;               // 0
    UCHAR  chScan;               // 1
    UCHAR  fbStatus;             // 2
    UCHAR  bNlsShift;            // 3
    USHORT fsState;              // 4
    ULONG  time;                 // 6    (32-bit offset is 8)
} KBDKEYINFO;

#pragma pack()

extern USHORT _Far16 _Pascal KBD16PEEK(KBDKEYINFO *,USHORT);

int main(void)
{
    KBDKEYINFO kbciKeyInfo;

    KBD16PEEK(&kbciKeyInfo, 0);

    return 0;

}
```

*Figure 3-29. Another Example of Using _Far16 _Pascal and _Seg16*

In this example, the packing #pragma is used. As previously discussed, this will force the IBM C Set/2 compiler to use WORD alignment for any structures defined in this range.

```
/*
 * 32-16 example 3.  Compile using the /Ss option.
 */

#include <os2.h>
#include <stdio.h>

#pragma pack(2)
typedef struct _VIOMODEINFO {    /* viomi */
    USHORT cb;
    UCHAR  fbType;
    UCHAR  color;
    USHORT col;
    USHORT row;
    USHORT hres;
    USHORT vres;
    UCHAR  fmt_ID;
    UCHAR  attrib;
    ULONG  buf_addr;
    ULONG  buf_length;
    ULONG  full_length;
    ULONG  partial_length;
    char * _Seg16 ext_data_addr;
} VIOMODEINFO;
#pragma pack()

#define VGMT_GRAPHICS                0x02

USHORT _Far16 _Pascal Vio16GetMode (VIOMODEINFO *, USHORT);

int main(void)
{
    VIOMODEINFO viomi;
    viomi.cb = sizeof(viomi);

    Vio16GetMode(&viomi, 0);

    printf("The current screen is %hd by %hd",viomi.col,viomi.row);

    return 0;
}
```

*Figure 3-30. Using the Packing Pragma Convention*

In Figure 3-30, note "char * _Seg16 ext_data_addr" in the structure definition. This structure is being passed into a 16-bit API. Therefore, any pointer that is passed in must be declared as _Seg16. This will insure that the pointer is stored in a selector:offset format, and will be usable by the target API.

## Function Calls to 16-Bit Modules

Thunking considerations affect the way in which a 16-bit function must be declared within the 32-bit module, and the way in which parameters passed to the 16-bit function are defined. When using IBM C Set/2, such functions and parameters must be declared using the *#pragma linkage* directive and the *_Far16 _Pascal* keyword, as shown in Figure 3-31.

```
#pragma stack16(8192)

USHORT _Far16 _Pascal MyFunction(USHORT FirstNum, HWND _Seg16 hWnd);
```

*Figure 3-31. Declaring a 16-Bit Function in 32-Bit Code*

Note the use of the *#pragma stack16* directive to set the stack size for all 16-bit function calls made from the 32-bit module.

Declaring a 16-bit function in this manner will cause the operating system to automatically perform thunking for all "value" parameters (that is, those other than pointers). Pointers passed as parameters must be explicitly defined using the *_Seg16* keyword, as shown in Figure 3-31.

### Using 16-Bit Window Procedures

A 32-bit application can access window procedures which reside in 16-bit modules, either statically linked or as DLLs. However, the differences between addressing methods requires some consideration on the part of the developer, since both the window handles and any pointers passed as message parameters will differ in their representation.

### Creating a Window

When a 32-bit application module creates a window, and the window procedure for that window resides in a 16-bit module (either statically linked or in a DLL), the calling routine must explicitly declare the 16-bit nature of the window procedure's entry point when registering the window class. This can become rather complex, since it involves invoking a 32-bit entry point from a 32-bit module, but passing a 16-bit entry point as a parameter.

A simpler solution is to build a registration routine within the 16-bit module, which registers the window class and creates the window. The 32-bit module then need only invoke this routine, and allow for the resulting 16-bit window handle. This technique has the added advantage that PM records the fact that the window was registered from a 16-bit module, and will automatically perform thunking for system-defined message classes. The technique is illustrated in Figure 3-32 on page 3-30.

Since the 16-bit module would typically be a DLL, the registration routine is declared in the 16-bit module as an exportable entry point using the EXPENTRY keyword.

The 32-bit module declares the registration routine *MakeMyWindow()* as a 16-bit function using the *Far16 _Pascal* keyword. Since the EXPENTRY keyword forces use of the pascal calling convention, the directive also specifies this calling convention. Note that if the registration routine and the window procedure were to reside in a DLL, this declaration would typically take place within a header file provided by the developer of the DLL.

The 32-bit module invokes the registration routine which registers the window class and creates the window. The registration routine then returns the window handle to the 32-bit module, which stores it in 16:16 format. Note that the registration routine in the 16-bit module is not aware that it is being called from a 32-bit module.

```
32-bit Module

#pragma stack16(8192)

HWND EXPENTRY16 MakeMyWindow(void);          /* 16-bit function prototype  */

HWND _Seg16 hWindow;                         /* 16:16 window handle        */
  :
  :
hWindow = MakeMyWindow();                    /* Call registration routine  */

16-bit Module

HWND EXPENTRY MakeMyWindow(void)             /* Registration routine       */
{
     HWND hCurrWindow;                       /* 16:16 window handle        */

     WinRegisterClass(...);                  /* Register window class      */
     hCurrWindow = WinCreateWindow(...);     /* Create window              */

     return(hCurrWindow);                    /* Return 16:16 window handle */
}
```

*Figure 3-32. Creating a 16-Bit Window From Within a 32-Bit Module*

This approach allows the same DLL to be accessed by both 16-bit and 32-bit applications concurrently. The developer of the DLL simply provides two separate header files containing declarations of the DLL's entry points, in the appropriate format for each programming environment.

## Passing Messages to 16-Bit Windows

Passing data between 16-bit and 32-bit window procedures via message parameters also requires consideration of the internal representations of the data types passed within the parameter. For system-defined message classes, this is handled automatically by OS/2 2.0, but for application-defined message classes the conversion between addressing methods must be handled by the application, since the operating system has no way of determining the intended contents of each parameter.

Simple "value" parameters (such as integers or characters) can be passed without the need for translation. Message parameters should be constructed using the standard PM macros.

When a pointer or handle is passed in a message parameter to a 16-bit window procedure, the pointer or handle must be translated to the 16:16 addressing method by the application. Since the 16-bit module is unlikely to have been written with code to achieve this conversion, it is the responsibility of the 32-bit module.

Conversion can be achieved using the *_Seg16* keyword to explicitly define a 16:16 pointer or handle, which is then placed in a message parameter using the MPFROMP macro. This is illustrated in Figure 3-33 on page 3-31. This example

shows the 32-bit code necessary to define and initialize a 16:16 pointer to be passed to a 16-bit window procedure.

```
typedef struct mystruct {                    /* Define data structure      */
                        CHAR * _Seg16 Name;
                        ULONG         ulA;
                        ULONG         ulB;
                        USHORT        usC;
                        } MYSTRUCT;

#pragma seg16(MYSTRUCT)                      /* Define pragma directive    */

MYSTRUCT * _Seg16 MyStruct;                  /* 16:16 pointer              */

BOOL              bSuccess;                  /* Success flag               */

MPARAM            lParam1;                   /* Message parameter          */

bSuccess = DosAllocMem(&MyStruct,            /* Allocate data structure    */
                       4096,                 /* Size of data structure     */
                       PAG_READ  |           /* Allow read access          */
                       PAG_WRITE |           /* Allow write access         */
                       PAG_COMMIT);          /* Commit storage immediately */

<Initialize structure if required>

lParam1 = MPFROMP(MyStruct);                 /* Set message parameter      */
```

*Figure 3-33. Passing a 16:16 Pointer as a Message Parameter*

The resulting message parameter can then be passed to a window in a 16-bit module using the normal WinPostMsg() or WinSendMsg() functions, using a 16:16 window handle obtained in the manner shown in Figure 3-32. Note that the data structure referenced by the pointer cannot be greater than 64KB in size, and must not cross a segment boundary. This is ensured in Figure 3-33 by using the *#pragma seg16* directive, since a structure defined using this pragma will automatically be aligned on a segment boundary.

## Passing Messages to 32-Bit Windows

The technique described above handles messages passed to a window in a 16-bit module. However, messages passed from that window to the 32-bit module may also require thunking. In order to perform this thunking, the 32-bit application can define a *thunk procedure* and register this procedure to PM, which then invokes the thunk procedure whenever a message is passed from within the window.

This registration is achieved using the WinSetWindowThunkProc() function, which is illustrated in Figure 3-34.

```
WinSetWindowThunkProc(hWindow,               /* Window handle              */
                      (PFN)ThunkProc16to32); /* Thunk proc entry point     */
```

*Figure 3-34. Mixed Model Programming—WinSetWindowThunkProc() Function*

The WinSetWindowThunkProc() function call is made from the 32-bit module. Since the window class for the window has been registered in the 16-bit module, PM recognizes that the thunk procedure is to handle 16-bit to 32-bit conversion.

A thunk procedure can be deregistered by issuing a WinSetWindowThunkProc() function call with the thunk procedure entry point address set to NULL.

Whenever PM invokes a thunk procedure for a message, it passes the normal four parameters accepted by a window procedure, along with the entry point address of the window procedure to which the message was to be passed. This can be the window procedure defined for the destination window when its class was registered, or a subclass window procedure defined by the application. Thus, thunking can take place irrespective of whether a window has been subclassed.

A sample thunk procedure is shown in Figure 3-35.

```
MRESULT ThunkProc16to32(HWND hwnd,                  /* Window handle      */
                        USHORT usMsg,               /* Message identifier */
                        MPARAM lParam1,             /* Message parameters */
                        MPARAM lParam2,
                        PFNWP wpWindow);            /* Window procedure   */
{
        switch (usMsg)
                {
                case WMP_MSG1:
                     lParam1=DosSelToFlat(lParam1); /* Thunk parameters    */
                     lParam2=DosSelToFlat(lParam2);
                     break;
                case WMP_MSG2:
                     lParam1=DosSelToFlat(lParam1); /* Thunk 1st parameter */
                     break;
                }
        return((*wpWindow)(hwnd,                    /* Call window proc   */
                           usMsg,
                           lParam1,
                           lParam2));
}
```

*Figure 3-35. Mixed Model Programming—Thunk Procedure*

The thunk procedure is invoked whenever a message is passed by the window in the 16-bit module to a window in the 32-bit module. The thunk procedure is similar in structure to a normal window procedure, but contains cases only for application-defined message classes, since thunking for system-defined message classes is performed by PM.

In Figure 3-35, the 16-bit window contains two application-defined message classes, WMP_MSG1 and WMP_MSG2. The first of these contains pointers in both parameters, and thus both parameters are thunked by the thunk procedure. The second message class contains a pointer in the first message parameter only; the second may contain an integer or some simple value parameter which does not require explicit thunking.

After performing the necessary thunking, the thunk procedure directly calls the window procedure entry point supplied by PM when the thunk procedure is invoked. Note that this is one of the few instances where direct invocation of a window procedure should be used; the correct sequence of message processing is preserved in this case because the thunk procedure itself is invoked either synchronously or asynchronously by PM, depending upon whether the message was sent or posted by the 16-bit window.

## Calling 32-Bit Code from 16-Bit Code

There is no mixed model support in the 16-bit tools. Hence, the 16-bit tools provide no support for flat code. This leaves the 32-bit module with all responsibilities for making the mixed model executable work correctly.

To call 32-bit routines from a 16-bit routine, it is necessary to declare the 32-bit routine with a 16-bit calling convention. It is also necessary for the 16-bit routine to get the address of the 32-bit routine in a 16:16 form. This is done by passing the address of the 32-bit routine to the 16-bit routine, and having the 16-bit routine call the 32-bit routine through a function pointer. The following code fragment illustrates this point:

```
/* 16.C */

void _Far16 _Pascal fn16 (void (* _Far _Pascal fnptr)(char *), char * text)
{
   fnptr (text);
   return;
}

/* 32.C */
#include <stdio.h>
void _Far16 _Pascal fn32 (char * text)
{
   printf ("%s\n",text);
   return;
}
void _Far16 _Pascal fn16 (void (* _Far16 _Pascal)(char *), char *);
int main (void)
{
   fn16 (fn32, "Hello, world");
   return 0;
}
```

*Figure 3-36. Calling 32-Bit Code from 16-Bit Code*

This example calls the fn16 routine, passing to it the address of the 32-bit routine fn32, plus a string to be printed. Fn16 calls fn32 through the function pointer fnptr, and fn32 converts its parameter to a flat pointer upon entry, switches into 32-bit mode, runs, and then switches back to 16-bit mode for the return to fn16.

This sample was written using IBM C Set/2 keywords. If you use a different compiler, use the keywords your compiler employs to achieve the same effect.

## Migrating to OS/2 2.0

If you have already made significant progress on a 16-bit OS/2 application, you can complete the application for that environment. Although your application will not take advantage of the features of 32-bit OS/2, it will run in that environment, allowing users to make use of the application until you complete your 32-bit revision.

If you are just beginning development of your application, you should consider developing both a 16-bit and 32-bit version of your application by using one of the following strategies:

1. **Write a 16-bit application but include 80386 code.**

   Gear the application for 16-bit OS/2, but include conditional 80386 code that can be used at run time if the software is run on a 80386 machine. (This can be useful for spreadsheet recalculations, for example, where the availability of the 80386 instruction set adds significantly to the application's performance.)

   **Advantages:** The application will be able to run under both 16- and 32-bit versions. In addition, it will offer improved performance transparently when run on a 80386 machine.

   **Disadvantages:** No linear memory management will be available. The application will be more difficult to port to other 32-bit flat model platforms. You will need to develop and support both 80386 and 80286 code in the application.

2. **Create conditional source code for 16- and 32-bit versions.**

   A single program source can be used with conditional compilation to produce executables for either a 16- or 32-bit format.

   **Advantages:** Provides maximum market coverage. The 80386 version can take advantage of the improved instruction set, linear memory, and so on.

   **Disadvantages:** Requires two sets of development files (include files and libraries) and both versions of a compiler and an assembler. Source code is larger and more difficult to maintain. Functions that take advantage of linear memory or other OS/2 2.0-specific functions must be written twice and conditionally compiled to create the appropriate version.

3. **Support 16- and 32-bit sources.**

   Develop and maintain both a 16-bit and a 32-bit version of the application.

   **Advantages:** Provides maximum market coverage. The 2.0 version has all the benefits of a 32-bit application.

   **Disadvantages:** Separate source code may make future changes easier, since a straightforward 80386 version already exists.

To guarantee unique names in the libraries, all 32-bit functions include the number "32" in the function name (for example, Dos32Open). In the header files, the 32-bit functions names are aliased so that you use only the standard name (for example, DosOpen) in your source file and let the compiler automatically replace it with the actual name. The compiler chooses the correct alias based on the development model selected.

## Summary

OS/2 2.0 supports four types of applications: full-screen, windowable, PM, and DOS/Windows. A full-screen application is any OS/2 application that does not create a PM message queue, and does not rely on the PM mouse and keyboard processing for input. A windowable application is a full-screen application that also can run in a window, or PM session. A PM application is any OS/2 application that creates a message queue. Generally, PM applications create one or more windows to interact with the user. A DOS/Windows application runs in an OS/2 DOS session in the protected, virtual 8086 mode of the 80386 microprocessor. A DOS/Windows application can be full-screen or windowed, and it can be run concurrently with other applications.

OS/2 applications can be further classified as pure 16-bit, mixed 16-bit, pure 32-bit, and mixed 32-bit applications. Pure 16-bit applications can be run under the 16-bit and 32-bit versions of the operating system, but cannot take advantage of the features of the 32-bit programming environment. Mixed 32-bit applications can only be run under the 32-bit version of the operating system. Like pure 16-bit OS/2 applications, they do not have access to the 32-bit virtual address space; however, because they have a 32-bit EXE format, they can take advantage of demand paging. Pure 32-bit applications incorporate the flat memory model and protection mechanisms which are common on a wide range of computer industry hardware platforms. They can only run under the 32-bit version of the operating system. Mixed 32-bit applications can run only under the 32-bit version of the operating system, although they can use 16-bit APIs. These applications can access the entire 32-bit virtual address space.

OS/2 2.0 provides different entry points for 16-bit and 32-bit APIs, making it possible to mix 16- and 32-bit code within a single EXE module. It is also possible to call 32-bit APIs from a 16-bit C program, and to call 16-bit APIs from a 32-bit C program. To support this, two different libraries—OS2286.LIB and OS2386.LIB— are supported and changes have been made to the include file architecture and to the compiler (such adding new keywords to support calling 16-bit functions).

In OS/2 2.0, the system supplies an interface between 16-bit and 32-bit code, called a *thunking layer*. The thunking layer translates calls to 16-bit functions into calls to 32-bit functions and calls to 32-bit functions into calls to 16-bit functions. In performing these translations, the thunk must take into account:

- The different memory models between the 16-bit version and 32-bit version of the operating system. The 16-bit segmented memory model needs to be mapped onto the 32-bit flat memory model in such a way that addresses can be quickly converted. This mapping is accomplished through a technique called *tiling*. The API functions, DosFlatToSel and DosSelToFlat, are used to convert 32-bit flat addresses to 16-bit segmented addresses and 16-bit segmented addresses to 32-bit flat addresses.
- Different parameter sizes
- 64K segment boundary problems
- Different call models

# Chapter 4. Comparison of 16-Bit and 32-Bit OS/2 Functions

This chapter describes the differences between 16-bit and 32-bit OS/2 functions for:

- Control Program
- Presentation Manager

**Note:** Specific uses of the OS/2 API functions are demonstrated in the Toolkit sample programs. For a complete listing of the API functions used by each sample program, see Appendix A, "Sample Programs Cross Index" on page A-1.

## Changes to the Control Program

Changes to Control Program functions are of two types: name changes and replacements or enhancements.

Many of the 16-bit function names were changed to be more consistent in the 32-bit release. The guidelines used to name functions are:

- Compliance with Get, Set, and Query semantics used in PM-SAA conventions
- Use of action verbs before nouns
- Use of consistent semantics for similar actions

File-system functions are significantly affected by name changes.

Some 16-bit functions have been completely redesigned for the 32-bit version of the operating system. This is particularly true in memory-management, semaphore, and signal functions.

Functions for the 32-bit operating system are described in the *OS/2 Version 2.0 Programming Tools and Information*. Some 16-bit functions are not supported by 32-bit code. These functions, though still supported by OS/2 2.0, are described in the *OS/2 2.0 Control Program Programming Reference*.

## Memory Management

The 16-bit version of the operating system uses a segmented-memory model. Applications can request segments as large as 64KB. They access data in those segments through pointers consisting of 16-bit selectors and offsets.

In the 32-bit OS/2 version of the operating system:

- Memory is requested by object, not by segment. Functions that allocate memory return 32-bit pointers to memory objects ranging in size from 1 page (page = 4K) to any size supported by available *swap space*, to a maximum of 448MB. Swap space refers to external storage devices used to store code and data which is not required for immediate processor execution.
- All pointer references are 32-bit near pointers. No segment register loads are involved; thus all of the segment registers are equal: DS = SS = ES.
- Virtual memory works by demand paging, rather than by compaction and segment swapping. This has important implications for sizing memory objects for optimum system performance.

All memory allocations in the 32-bit version of the operating system, whether private or shared, initialize committed pages to all zeros. Programs can rely on this when determining the initial contents of memory.

Table 4-1 on page 4-2 summarizes the 16-bit and corresponding 32-bit memory-management functions.

*Table 4-1. 16-Bit to 32-Bit Memory-Management Functions*

| 16-Bit Name | 32-Bit Name |
|---|---|
| DosAllocSeg | N/A |
| DosAllocShrSeg | N/A |
| DosGetShrSeg | N/A |
| DosGetSeg | N/A |
| DosGiveSeg | N/A |
| DosReallocSeg | N/A |
| DosFreeSeg | N/A |
| DosAllocHuge | N/A |
| DosGetHugeShift | N/A |
| DosReallocHuge | N/A |
| DosCreateCSAlias | N/A |
| DosLockSeg | N/A |
| DosUnlockSeg | N/A |
| DosMemAvail | N/A |
| DosSizeSeg | N/A |
| DosSubAllocMem | DosSubAllocMem |
| DosSubFreeMem | DosSubFreeMem |
| DosSubSetMem | DosSubSetMem |
| N/A | DosSubUnsetMem |
| N/A | DosAllocMem |
| N/A | DosAllocSharedMem |
| N/A | DosGetNamedSharedMem |
| N/A | DosGetSharedMem |
| N/A | DosGiveSharedMem |
| N/A | DosFreeMem |
| N/A | DosSetMem |
| N/A | DosQueryMem |

## Allocating Memory

| Version | Functions |
|---|---|
| 16-bit | DosAllocSeg, DosAllocHuge |
| 32-bit | DosAllocMem |

The 32-bit function replaces both 16-bit functions. The new allocation scheme has several concepts specific to the paged environment. Among them is the ability to control access to the memory object and to control when to commit the pages.

The example shown in Figure 4-1 illustrates a typical request for 68KB of memory under both the 16-bit and 32-bit versions of the operating system.

```
16-Bit:

 SEL sel1, sel2;
 PBYTE pabObject1, pabObject2;
 USHORT offSelector, usShiftCount;

 /* Method 1 -- allocate two selectors. */

 DosAllocSeg(0, &sel1, SEG_NONSHARED);
 DosAllocSeg(4096, &sel2, SEG_NONSHARED);
 pabObject1 = MAKEP(sel1, 0);
 pabObject2 = MAKEP(sel2, 0);

 /* Code must use pabObject2 for memory references above 64KB. */


 /* Method 2 -- use DosAllocHuge. */

 DosGetHugeShift(&usShiftCount);
 offSelector = 1 << usShiftCount;
 DosAllocHuge(1, 4096, &sel1, 2, SEG_NONSHARED);
 pabObject1 = MAKEP(sel1, 0);
 pabObject2 = MAKEP(sel1 + offSelector, 0);

 /* Code can use pabObject1 and pabObject2 as above, or take advantage of
  * the arrangement of the huge selectors by using pointers of the type
  * pab = MAKEP(sel1 + offset >> 16, offset / 16). */

32-Bit:

 PBYTE pabObject;

 DosAllocMem(&pabObject, 69632, PAG_COMMIT | PAG_READ | PAG_WRITE);
```

*Figure 4-1. Allocating Memory*

Both the 16-bit and 32-bit code shown allocate 69632 bytes, returning a pointer to the first byte in pabObject (or pabObject1). The memory has both read and write access, the typical permission for dynamically allocated memory objects. In addition, the pages are committed when the call returns, and storage space for the objects is reserved; the program can now make memory references for reading or writing to the range, pabObject through pabOjbect plus 69631. Under the segmented architecture, programs cannot read past the first 64KB without changing the selector. In the 32-bit version, the entire range of addresses corresponding to the object is directly accessible.

In the preceding example, note that the paged memory requested has a *granularity* of 4KB; exactly 68KB has been requested. Granularity is the unit of memory allocation specific to an operating system and hardware platform. The granularity is 4KB for OS/2 2.0 and 16 bytes to 64000 bytes for 16-bit versions of the operating system. Had 68KB plus 1 byte been requested, the system would have to commit 72KB of memory; the extra single byte would use up 4KB of the linear address space. Although pages of memory will be swapped out as needed, directly allocating many small objects with the DosAllocMem function is inefficient. Instead,

use the DosSubAllocMem function or the C library malloc routines. (Malloc is a standard C generic method which translates a programmer's request for memory allocation to the specific operating system implementation of memory allocation. This technique removes some operating system dependent code from the C programmer.) You also can implement your own *heap* by requesting a large, page-aligned block and subdividing it as needed by your code. A heap is a large, allocated memory region that is subdivided for private use. When writing applications expected to run under the 32-bit and earlier versions, remember that the DosAllocSeg function will not pack segments into the same page. Segment sizes of 16KB to 32KB are a reasonable compromise for code that must run under either system. (Larger segments become more efficient under paging, but less efficient under segmentation.)

## Freeing Memory

| Version | Functions |
|---|---|
| **16-bit** | DosFreeSeg |
| **32-bit** | DosFreeMem |

These functions release previously allocated, private or shared, memory objects. For DosFreeMem, the address should always be the base address returned by a call to the DosAllocMem function. With shared memory, the associated reference count is decremented by one; when all holders have released the shared memory, the system will reclaim the space. The DosFreeSeg function also is used to release selectors granted for huge memory and code aliasing (a feature of 286 architecture and OS/2 Version 1.X implementation which allows multiple selectors to reference the same code segment), which are not needed under under the 32-bit version.

The example in Figure 4-2 shows how to release memory under both 16-bit and 32-bit versions.

```
16-Bit:
 SEL sel;
 DosFreeSeg(sel);
32-bit:
 PBYTE pabObject;
 DosFreeMem(pabObject);
```

*Figure 4-2. Freeing Memory*

## Suballocating Memory

| Version | Functions |
|---|---|
| **16-bit** | DosSubSet, DosSubAlloc, DosSubFree |
| **32-bit** | DosSubSetMem, DosSubAllocMem, DosSubFreeMem, DosSubUnsetMem |

Suballocation enables large memory object to be subdivided. DosSubAllocMem and DosSubFreeMem allow a simple heap arrangement, with low performance requirements (few instructions required to perform repetitive allocating and freeing

of small memory blocks), that is sufficient for many memory operations. This allocation package normally will serialize requests made by different threads within a process (when using WinAllocMem, the application must provide for synchronization). Table 4-2 shows the differences for the DosSubSetMem function.

*Table 4-2. 16-Bit and 32-Bit DosSubSetMem*

| Feature | 16-Bit | 32-Bit |
|---|---|---|
| Granularity of size for DosSubSetMem and DosSubAllocMem | 4 bytes | 8 bytes |
| Minimum size for DosSubSetMem | 12 bytes: 8-byte header plus 4-byte allocation region. | 40 bytes: 32-byte header plus 8-byte allocation region. |
| When DosSubSetMem = 0 | A request of 64KB. | A request of 4GB (GB equals 1 073 741 824 bytes). (**this call will fail due to not enough memory **) |
| Maximum size of request for DosSubAllocMem | Segment size: 12 bytes (reserved for header). | Object size: 32 bytes (reserved for header). |
| Serialization of requests | Always. | Selectable. |
| Additional flags | None. | The requestor wants a suballocating function to manage the commitment of the pages spanned by the memory pool; the requestor requires access to the memory pool to be serialized. |

In both 16-bit and 32-bit versions, the maximum size for DosSubSetMem is the size of the memory object being used for the heap. Because memory requests may be larger than 64KB in the 32-bit version, the new version of DosSubSetMem provides more flexibility in choosing the size of the heap, and removes any external barriers to size limits for data objects within the heap. Also, both versions initially allow the size requested in DosSubSetMem to be less than the size of the segment (16-bit version) or object (32-bit version). The suballocator then uses the low area of the segment (object) for its allocation region. If DosSubSetMem later is used to make the allocation region larger, the memory currently in use (including the organization information in the header) is guaranteed to remain valid. For all versions, DosSubFreeMem must include the size of the object being freed; the suballocator does not maintain the size of each object, and space that is not freed remains locked.

The example in Figure 4-3 demonstrates the suballocation and freeing of 16 bytes from a 4KB memory object.

```
16-Bit:

SEL sel;
USHORT usOffset;
PBYTE pabSmallObject;

DosAllocSeg(4096, &sel, SEG_NONSHARED);
DosSubSet(sel, 1, 4096);    /* uses entire region for suballocation */

DosSubAlloc(sel, &usOffset, 16);              /* requests 16 bytes */
pabSmallObject = MAKEP (sel, usOffset);

    .
    .
    .

DosSubFree(sel, usOffset, 16);    /* frees all 16 bytes */

32-Bit:

PBYTE pabObject, pabSmallObject;

DosAllocMem(&pabObject, 4096, PAG_COMMIT | PAG_READ | PAG_WRITE);
DosSubSetMem(pabObject, 1, 4096);/* uses entire region for suballocation */

DosSubAllocMem(pabObject, &pabSmallObject, 16); /* requests 16 bytes    */

    .
    .
    .

DosSubFreeMem(pabObject, pabSmallObject, 16);   /* frees all 16 bytes    */
```

*Figure 4-3. Suballocating Memory*

## Using Named Shared Memory

| Version | Functions |
|---|---|
| **16-bit** | DosAllocShrSeg, DosGetShrSeg |
| **32-bit** | DosAllocSharedMem, DosGetNamedSharedMem |

Named shared memory enables any process that specifies the correct name to gain access to the shared segment. One process must allocate the shared segment initially; others then can gain access by using the DosGetShrSeg function. Named shared memory uses the file-system name space; all names must be of the form, \sharemem\name, where *name* represents a valid file name. The system will not allow subsequent calls that attempt to allocate named memory to a name already in use.

The example in Figure 4-4 demonstrates the allocation and subsequent sharing of a named segment.

```
16-bit:

SEL sel;

/* Process 1. */

DosAllocShrSeg(4096, "\\sharemem\\mystuff", &sel);


/* Process 2 must come AFTER segment has been allocated by process 1. */

DosGetShrSeg("\\sharemem\\mystuff", &sel);

32-bit:

PBYTE pabSharedObject;

/* Process 1. */

DosAllocSharedMem(&pabSharedObject, "\\sharemem\\mystuff", 4096,
    PAG_COMMIT | PAG_READ | PAG_WRITE);


/* Process 2 must come AFTER memory has been allocated by process 1. */

DosGetNamedSharedMem(&pabSharedObject, "\\sharemem\\mystuff",
    PAG_READ | PAG_WRITE);
```

*Figure 4-4. Using Named Shared Memory*

Both processes can read from and write to the shared segment. The selectors returned to the cooperating processes in the 16-bit version will refer to the same virtual-memory segment. The pointers returned in the 32-bit version are guaranteed to be identical for identical names; that is, the shared memory lies at the same virtual address in each process.

## Using Unnamed Shared Memory

| Version | Functions |
|---|---|
| 16-bit | DosGiveSeg, DosGetSeg |
| 32-bit | DosGiveSharedMem, DosGetSharedMem |

Unnamed shared memory requires that the process allocating the memory object transfer a handle to it to another process. (The initial process must know the process identifier of the recipient.) Unnamed shared memory does not communicate through the file-system name space.

The example in Figure 4-5 demonstrates how a memory segment is allocated.

```
16-Bit:

SEL sel, selRecipient;
PID pidRecipient;

DosAllocSeg(4096, &sel, SEG_GIVEABLE);
DosGiveSeg(sel, pidRecipient, &selRecipient);

/* Use some IPC mechanism to transmit selRecipient. */

32-Bit:

PBYTE pabSharedObject;
PID pidRecipient;

DosAllocSharedMem(&pabSharedObject, (PSZ) NULL, 4096,
    PAG_COMMIT | OBJ_GIVEABLE | PAG_READ | PAG_WRITE);
DosGiveSharedMem(pabSharedObject, pidRecipient, PAG_READ | PAG_WRITE);

/* Use some IPC mechanism to transmit pabSharedObject. */
```

*Figure 4-5. Giving Unnamed Shared Memory*

The example in Figure 4-6 demonstrates how a segment is obtained.

```
16-Bit:

/* Process 1. */
SEL selShared;

DosAllocSeg(4096, &selShared, SEG_GETTABLE);

/* Use some IPC mechanism to transmit selShared. */

/* Process 2, receive selShared via IPC mechanism. */
SEL sel;

DosGetSeg(sel);

32-Bit:

/* Process 1. */
PBYTE pabSharedObject;

DosAllocSharedMem(&pabSharedObject, (PSZ) NULL, 4096,
    PAG_COMMIT | OBJ_GETABLE | PAG_READ | PAG_WRITE);

/* Use some IPC mechanism to transmit pabSharedObject. */

/* Process 2, receive pabSharedObject via IPC mechanism .*/
PBYTE pabSharedObject;

DosGetSharedMem(pabSharedObject, PAG_READ | PAG_WRITE);
```

*Figure 4-6. Getting Unnamed Shared Memory*

## Generating Dynamic Code

| Version | Functions |
|---|---|
| **16-bit** | DosCreateCSAlias |
| **32-bit** | DosAllocMem |

Before the 32-bit version of the operating system, programs requiring that code be generated during execution used the DosCreateCSAlias function. A data segment would be used to write the operation codes (opcodes), then DosCreateCSAlias would return a new code segment, which could be used to make a call to the dynamically compiled code. With the 32-bit version of the operating system, the DosAllocMem function, specifying PAG_WRITE | PAG_EXECUTE access, is used for writing in the memory object and for making it executable.

The example in Figure 4-7 demonstrates the generation and execution of program-compiled code.

**16-Bit**

```
SEL selData, selCode;
LONG (PASCAL FAR *SampleFunction) (PSZ pszParam1, USHORT usParam2);
LONG lReturn;
PBYTE pabFunction;

DosAllocSeg(4096, &selData, SEG_NONSHARED);
pabFunction = MAKEP(selData, 0);

/* Write program into memory by using pabFunction. */
      .
      .
/* Alias code selector to selData. */

DosCreateCSAlias(selData, &selCode);

/* Free data selector. Only needed to write program. */

DosFreeSeg(selData);
SampleFunction = (LONG (PASCAL FAR *)(PSZ, USHORT))(MAKEP(selCode, 0));

lReturn = (*SampleFunction)("Hello", 5);

/* Done with code selector. */

DosFreeSeg(selCode);
```

**32-Bit:**

```
PBYTE pabFunction;
LONG (*SampleFunction) (PSZ pszParam1, USHORT usParam2);
LONG lReturn;

/* ... */

DosAllocMem(&pabFunction, 4096,
    PAG_COMMIT | PAG_WRITE | PAG_EXECUTE);
   .
   . /* Write program into memory by using pabFunction. */
   .

SampleFunction = (LONG (*)(PSZ, USHORT))pabFunction;
lReturn = (*SampleFunction)("Hello", 5);

/* Done with code memory object. */

DosFreeMem(pabFunction);
```

*Figure 4-7. Generating Dynamic Code*

## Determining Available Memory

| Version | Functions |
|---|---|
| **16-bit** | DosMemAvail |
| **32-bit** | No equivalent |

The DosMemAvail function was used in earlier versions of the operating system to return the largest consecutive block of free memory available without resorting to the swapping, moving, or discarding of segments. Because this function is not serialized with system functions, the value returned will only be accurate *when* it is determined by the function, and not necessarily after the system returns to the issuing code. Therefore, the only practical use for the function is to measure system-memory load. No corresponding function exists in the current 32-bit version.

## Discarding Memory Objects

| Version | Functions |
|---|---|
| **16-bit** | DosLockSeg, DosUnlockSeg |
| **32-bit** | No equivalent |

Earlier segmented versions of the operating system allowed the discarding of segments as one means of managing virtual memory. Information that a program could easily regenerate could be stored in a discardable segment, rather than being swapped to disk. The 32-bit version of the operating system uses least-recently-used swapping for its *memory-overcommitment* feature and does not support the discarding of pages. Memory overcommitment is the ratio between the amount of code and data which the operating system is capable of executing or referencing and the available physical storage. If the available physical storage is less than the amount of code and data, the difference is kept on the swap space. This may lead to larger swap-space requirements for programs that make extensive use of discardable objects. The DosLockSeg and DosUnlockSeg functions have no counterparts in the 32-bit system, and the allocation flag, `SEG_DISCARDABLE`, is not used.

## Setting Memory Commitment and Access

| Version | Functions |
|---|---|
| **16-bit** | No equivalent |
| **32-bit** | DosSetMem |

Pages that are not committed at allocation time can be selectively committed with the DosSetMem function. This function also can be used to change the access type of committed pages of memory. These functions have no counterparts in earlier versions of the operating system.

## Checking a Process's Virtual-Memory Map

| Version | Functions |
|---|---|
| **16-bit** | No equivalent |
| **32-bit** | DosQueryMem |

Any thread can identify the current status of the memory map for the corresponding process. The DosQueryMem function can be called to determine the access, commitment, and private or shared status of blocks of memory. It is the only function that allows memory addresses outside the range currently allocated to the

process. There is no corresponding function in earlier versions of the operating system.

## Threads and Processes

OS/2 2.0 supports multiple processes, as well as multiple threads within each process. The functions supporting process and thread control are very similar in both the 16-bit and 32-bit versions; however, small differences do exist for some functions. Table 4-3 shows the 16-bit and corresponding 32-bit functions that support process and thread control.

*Table 4-3. 16-Bit to 32-Bit Tasking Functions*

| 16-Bit Name | 32-Bit Name |
|---|---|
| DosCreateThread | DosCreateThread |
| DosCWait | DosWaitChild |
| N/A | DosWaitThread |
| DosEnterCritSec | DosEnterCritSec |
| DosExecPgm | DosExecPgm |
| DosExit | DosExit |
| DosExitCritSec | DosExitCritSec |
| DosExitList | DosExitList |
| DosGetInfoSeg | N/A |
| DosGetEnv | DosGetInfoBlocks |
| DosGetPrty | N/A |
| DosKillProcess | DosKillProcess |
| DosSetPrty | DosSetPriority |
| DosGetPID | N/A |
| DosGetPPID | N/A |
| DosR2StackRealloc | N/A |
| DosCallBack | N/A |
| DosRetForward | N/A |
| N/A | DosDebug |
| DosPTrace | N/A |
| DosResumeThread | DosResumeThread |
| DosSuspendThread | DosSuspendThread |
| DosQSysInfo | DosQuerySysInfo |

### Creating Threads

| Version | Functions |
|---|---|
| 16-bit | DosCreateThread |
| 32-bit | DosCreateThread |

The 32-bit version of the DosCreateThread function has several new features:

- The function pointer is now a near pointer. Data declared with a near pointer resides in the default data segment and is referenced with 32-bit addresses.

- Clearing the ES register is no longer required upon entry to the new thread. Because all threads in a process have ES = DS = SS, memory violations from accessing a segment cleared by another thread do not occur.
- A single doubleword argument can be passed to the newly created thread through the creation function. This often will be the address of the information required by the new thread.
- The thread can be started in either the active or the suspended state. If it is started in the suspended state, another thread should eventually call the DosResumeThread function to activate it.
- Stack memory is now allocated by the system. The system will first round this size up to the nearest page-size boundary (4KB in 32-bit version) and then use that number to allocate the entire range of stack memory. Memory will be committed dynamically for the thread's stack, using the guard-page feature of the 80386 microprocessor.

  In normal operation, memory requests will gradually fill up the current page until a new request raises a guard-page-entered exception. The system will then commit the old guard page and move it down. Because only one guard page is used, allocations from the stack greater than 4KB for any thread (even the initial one) should be handled by a compiler-generated stack probe.

## Controlling Threads

| Version | Functions |
|---|---|
| **16-bit** | DosSuspendThread, DosResumeThread |
| **32-bit** | DosSuspendThread, DosResumeThread |

These functions perform the same actions. DosResumeThread also can be used to start execution in threads that were initially created in the suspended state with DosCreateThread.

## Exiting from Threads and Processes

| Version | Functions |
|---|---|
| **16-bit** | DosExit |
| **32-bit** | DosExit |

DosExit can be used to end the entire process, or a single thread within the process. There are several important points to remember regarding its use:

- The system can create threads for the process, independent of the threads created by the application. Therefore, to exit from the process, always use EXIT_PROCESS, rather than trying to exit from or to end each thread.
- In the 32-bit version of the operating system, ending thread 1 (the initial thread for the process) ends the entire process. It is equivalent to calling DosExit with EXIT_PROCESS as the first parameter.
- When a process is ending, one thread will be kept active and used to execute the exit list for the process. When the execution of the list is completed, the system will free all resources held by the process, and the process will end. The result code (exit code) will be returned to the parent process.

## Ending Other Processes

| Version | Functions |
|---|---|
| **16-bit** | DosKillProcess |
| **32-bit** | DosKillProcess |

In both the 16-bit and 32-bit versions of the operating system, a process can specify tree termination (ending another process and all its descendants) only if the target is the process itself, or a process begun by it with the DosExecPgm function and an AsyncTraceFlags of 2 (EXEC_ASYNCRESULT). If the target process has finished, its child processes will still be ended.

In all versions, the system erases the contents of system buffers; if internal buffers exist (such as C file buffers), the process must use an exception handler to trap the termination signal, end its functions in an orderly manner, and then exit.

## Handling Critical Sections

| Version | Functions |
|---|---|
| **16-bit** | DosEnterCritSec, DosExitCritSec |
| **32-bit** | DosEnterCritSec, DosExitCritSec |

These functions operate the same in both the 16-bit and 32-bit versions of the operating system. The DosEnterCritSec function can generate an overflow error if the count should exceed 65535; DosExitCritSec can generate an underflow error if the count becomes less than 0. In either case, the function will not have taken effect. In all versions of the operating system, the thread-handling signals can still be activated in a critical section and should be written in such a way that it will not interfere with critical resources.

## Waiting for Threads

| Version | Functions |
|---|---|
| **16-bit** | DosCWait |
| **32-bit** | DosWaitChild, DosWaitThread |

DosCWait and DosWaitChild operate the same in both the 16-bit and 32-bit versions of the operating system. DosWaitThread is new for the 32-bit version of the operating system. It enables a thread to wait for another thread within the process to end. A thread can wait for any thread, or for a specific thread, to terminate.

## Getting Thread and Process Information

| Version | Functions |
|---|---|
| **16-bit** | DosGetEnv, DosGetInfoSeg, DosGetPrty, DosGetPID, DosGetPPID |
| **32-bit** | DosGetInfoBlocks, DosQuerySysInfo |

Information services have changed slightly in the 32-bit version of the operating system. Values that are constants are all accessible with DosQuerySysInfo. Other information functions (such as DosGetDateTime) must be used to retrieve dynamic, system-wide information.

Per-thread and per-process information is now retrieved through DosGetInfoBlocks, which returns the address of the Thread Information Block (TIB) of the current thread and the address of the Process Information Block (PIB) of the current process.

Several items of the TIB are kept in a read/write area of the process address space. Each data item is a doubleword field that describes the current thread as follows:

```
Exception List Chain     /* Current-thread identifier                  */
Base ESP                 /* Address of the base of the thread stack    */
Thread Stack Limit       /* Address of the end of the thread stack     */

System-specific Thread   /* Address of a thread information block      */
Information              /* that is specific to an operating system    */

Version                  /* Version number of the TIB                  */
User Data                /* Unused field that is available for use by */
                         /* the application                            */
```

The system-specific TIB contains the following doubleword fields:

```
TID                      /* Current-thread identifier                  */
Priority                 /* Current-thread priority                    */
Version                  /* Version number of the system-specific TIB */

Must Complete            /* The low-order word maintains a count for   */
                         /* DosEnterMustComplete and                   */
                         /* DosExitMustComplete.  The high-order       */
                         /* word is reserved                           */
```

Several items of the PIB are kept in a read/write area of the process address space. Each data item is a doubleword field that describes the current process as follows:

```
PID                      /* Current-process identifier                 */
PPID                     /* Parent-process identifier                  */
Module Handle            /* Module handle of the current process       */
Command Line             /* Address of the command line                */
Environment              /* Address of the environment block           */
Status                   /* Status of the current process              */
Type                     /* Type of the current process                */
```

Information that is specified for the "current thread" refers to the thread calling DosGetInfoBlocks. Process information is shared by all threads in a process.

This information replaces the function of DosGetPrty, DosGetPID, and DosGetPPID, and part of DosGetInfoSeg.

The example in Figure 4-8 on page 4-16 determines the process identifier, parent-process identifier, thread priority, and current hour.

```
16-Bit:

PIDINFO pidi;
SEL selGlobalSeg, selLocalSeg;
PGINFOSEG pgis;
PID pid, pidParent;
USHORT usPriority;
UCHAR uchHour;

DosGetPID(&pidi);
pid = pidi.pid;
pidParent = pidi.pidParent;

DosGetPrty(PRTYS_THREAD, &usPriority, 0);

DosGetInfoSeg(&selGlobalSeg, &selLocalSeg);
pgis = MAKEPGINFOSEG(selGlobalSeg);
uchHour = pgis->hour;

32-Bit:

DATETIME dateTime;
PTIB ptib;
PPIB ppib;
PID pid, pidParent;
USHORT usPriority;
UCHAR uchHour;

DosGetInfoBlocks(&ptib, &ppib);
pid = ppib->pib_ulpid;
pidParent = ppib->pib_ulppid;
usPriority = ptib->tib_ulpri;

DosGetDateTime(&dateTime);
uchHour = dateTime.hours;
```

*Figure 4-8. Getting Thread and Process Information*

## Starting Programs

| Version | Functions |
|---|---|
| **16-bit** | DosExecPgm |
| **32-bit** | DosExecPgm |

These functions perform the same action.

## Debugging Programs

| Version | Functions |
|---|---|
| **16-bit** | DosPtrace |
| **32-bit** | DosDebug |

The DosDebug function is new for the 32-bit version of the operating system. It provides 32-bit debugger support for 16-bit and 32-bit applications. The DosPtrace function is retained to provide debugging support for 16-bit debuggers. Due to the equivalencies between DosPtrace and DosDebug, DosPtrace can be implemented

on top of DosDebug through a thunk layer. This thunk layer is needed to convert between DosPtrace 16:16 addresses and DosDebug 0:32 addresses, between DosPtrace word fields and DosDebug doubleword fields, and so on.

## 16-Bit Functions with No 32-Bit Counterparts

The DosR2StackRealloc and DosCallBack functions exist in previous versions of OS/2 2.0 because of the three-ring nature of the 80286 privilege levels. The linear addressing method used in OS/2 2.0 supports a two-level (supervisor/user) privilege mechanism, which makes these functions unnecessary.

## Semaphores

The 16-bit semaphore functions have the following shortcomings:

- There is a possibility of missing "clear" events.
- The individual semaphore state is not adequately cleared by the system at process or thread termination.
- The system limit on the total number of semaphores is too restrictive.
- Thread termination cannot be detected in a critical section, increasing the possibility that resources can be left in an unresolved state.
- There is a possibility of a *spurious wake-up*. A spurious wake-up results when code that has suspended execution (due to a semaphore) resumes execution before the semaphore is freed by another thread or process.
- Semaphore usage is overloaded, because the same semaphore mechanism is used for both signaling and mutual exclusion.
- The ability of any thread to determine the state of a semaphore, other than the thread that owns the semaphore, is limited.

The 32-bit versions of the semaphore functions address these problems, and also attempt to provide the speed of the former "fast-safe RAM" semaphores. (Recall that for fast-safe RAM, serialization is controlled by the system and contained in application memory). None of the 32-bit semaphore functions are compatible with the 16-bit versions.

There are two classes of 32-bit semaphores: *private* and *shared*. A process can have up to 64K of private semaphores, available only to threads within that process, and can also access up to 64K of shared semaphores, available to all processes in the system. In addition, there are three types of 32-bit semaphores: mutex, event, and muxwait.

| Type | Description |
|---|---|
| **Mutex** | **(Mutual Exclusion)**: Used by several threads within a process, or by several processes, to protect access to a critical region. A typical use would be to prevent more than one thread at a time from updating a file on disk. |
| **Event** | Provide a signaling mechanism among threads or among several processes. A typical use would be to manage shared memory: Process 1 writes to the shared region then uses an event semaphore to signal processes 2 and 3 that they can access the shared data. |

**Muxwait** (**Multiple Wait**): Enable a thread to wait on several event or mutex semaphores simultaneously. It is a compound semaphore that consists of up to 64 event semaphores or mutex semaphores (the two types cannot be mixed). A typical use would be when a thread requires access to several shared regions of memory at once. The system blocks the thread until the thread acquires ownership of all mutex semaphores protecting the shared regions. The thread can then access the regions. Meanwhile, the system prevents access by other threads that are using the muxwait semaphore.

## Using Semaphores

Several general rules apply when using the new 32-bit semaphore functions.

Semaphores can be created as either named or unnamed entities. If the name parameter is other than NULL, it must be a name of the form `\sem32\name`, where *name* adheres to the file-system naming conventions. No record of the semaphore is actually kept on the disk; when the last process closes a named shared semaphore, the semaphore is removed from the system. A named semaphore is always shared (available to other processes that know the name). If the name field is NULL, an unnamed semaphore is created. Unnamed semaphores can be either private to a process or shared among processes, depending on whether the DC_SEM_SHARED flag is set when the semaphore is created. Semaphores intended for use solely among threads of the same process should be made private.

The sequence for using shared semaphores among processes is as follows:

1. Process 1 calls the appropriate function to create the semaphore, initializing it through the creation parameters.
2. Processes 2 through *n* use the same name (named semaphores) or receive the same semaphore handle (shared unnamed semaphores), and each calls the appropriate function to obtain access to the semaphore.
3. The processes use the semaphore.
4. Each process calls the appropriate closing function (DosCloseMutexSem, DosCloseEventSem, DosCloseMutexWaitSem) when finished with the semaphore. The system allows nested Open and Close functions for semaphores, up to 64KB deep (65536 simultaneous open functions); however, to release a semaphore from a process, the Close function must be executed as many times as the Open function was executed. When all processes have closed the semaphore, the system frees the associated name (named semaphores) or handle (unnamed semaphores).

If a thread ends while owning a mutex semaphore, future request or query function calls will return the error value, ERROR_SEM_OWNER_DIED. The mutex semaphore then must be closed, because its use is no longer valid. If ERROR_SEM_OWNER_DIED was returned from a muxwait semaphore, all semaphores in the group should be queried and those whose owners have ended must be closed. They also should be removed from the muxwait group. Event semaphores (and muxwait semaphores involving event semaphores) will not receive this error if the process that expected to call DosPostEventSem ends. Processes waiting for untrustworthy events should therefore use the time-out fields when waiting to periodically re-evaluate the situation. For example, process 1 waits for an event semaphore that process 2 is supposed to post. Process 2 may fail to perform the post due to a large number of environment factors. Therefore process 1 should set a time-out in case process 2 cannot perform its function.

Table 4-4 on page 4-19 summarize the 16-bit and corresponding 32-bit semaphore functions.

*Table 4-4. 16-Bit to 32-Bit Semaphore Functions*

| 16-Bit Name | 32-Bit Name |
|---|---|
| DosSemClear | N/A |
| DosSemRequest | N/A |
| DosSemSet | N/A |
| DosSemSetWait | N/A |
| DosSemWait | N/A |
| DosMuxSemWait | N/A |
| DosCloseSem | N/A |
| DosCreateSem | N/A |
| DosOpenSem | N/A |
| DosFSRamSemRequest | N/A |
| DosFSRamSemClear | N/A |
| N/A | DosCreateMutexSem |
| N/A | DosOpenMutexSem |
| N/A | DosCloseMutexSem |
| N/A | DosRequestMutexSem |
| N/A | DosReleaseMutexSem |
| N/A | DosQueryMutexSem |
| N/A | DosCreateEventSem |
| N/A | DosOpenEventSem |
| N/A | DosCloseEventSem |
| N/A | DosResetEventSem |
| N/A | DosPostEventSem |
| N/A | DosWaitEventSem |
| N/A | DosQueryEventSem |
| N/A | DosCreateMuxWaitSem |
| N/A | DosOpenMuxWaitSem |
| N/A | DosCloseMuxWaitSem |
| N/A | DosWaitMuxWaitSem |
| N/A | DosAddMuxWaitSem |
| N/A | DosDeleteMuxWaitSem |
| N/A | DosQueryMuxWaitSem |

## Signalling Events with Semaphores

| Version | Functions |
|---|---|
| 32-bit | DosCreateEventSem, DosOpenEventSem, DosCloseEventSem, DosResetEventSem, DosPostEventSem, DosWaitEventSem, DosQueryEventSem |

Event semaphores have two basic states—*posted* and *reset*. The DosPostEventSem function posts the semaphore if it is not posted, and increments the number of postings (until the number reaches the maximum). All threads that have used DosWaitEventSem on this semaphore are tagged for wake-up (to resume execution).

The DosResetEventSem function resets the semaphore, if it is not already reset, and blocks all threads that use the DosWaitEventSem function afterwards. The DosQueryEventSem function returns the current number of postings to the semaphore. A count of 0 indicates that the semaphore is in the reset state, and threads that wait on the semaphore are blocked.

The examples in Figure 4-9 and Figure 4-10 on page 4-21 demonstrate the use of event semaphores. A single process (process 1) writes to a shared memory region. Several processes (2 through *n*) then read the data for further processing.

```
Process 1:

HSYSSEM hssm;

DosCreateSem(CSEM_PRIVATE, &hssm, "\\sem\\mysignal");

/* Set the semaphore. */

DosSemSet(hssm);

/* Fill shared memory region with data. */

/* Signal that data is ready. */

DosSemClear(hssm);

/* Program continues. Other processes read data and finish with the
 * event semaphore.*/

DosCloseSem(hssm);

All other processes:

HSYSSEM hssm;

DosOpenSem(&hssm, "\\sem\\mysignal");
DosSemWait(hssm, SEM_INDEFINITE_WAIT);

/* Use shared memory data now. */

DosCloseSem(hssm);
```

*Figure 4-9. Using Event Semaphores—16-Bit Version*

```
Process 1:

HEV hev = 0;    /* handle to event semaphore */

DosCreateEventSem("\\sem32\\mysignal", &hev, 0, FALSE);

/* Fill shared memory region with data. */
   .
   .
DosPostEventSem(hev);

/*
 * Program continues. Other processes read data and finish with
 * the event semaphore.
 */

DosCloseEventSem(hev);

All other processes:

/* Handle to event semaphore */
HEV hev = 0;

DosOpenEventSem("\\sem32\\mysignal", &hev);
DosWaitEventSem(hev, SEM_INDEFINITE_WAIT);

/* Use shared memory data now. */
   .
   .
DosCloseEventSem (hev);
```

*Figure 4-10. Using Event Semaphores—32-Bit Version*

## Using Event Semaphores Between 16- and 32-bit Code

New 32-bit applications can pass a 32-bit event semaphore handle to 16-bit code. This handle can then be used by the 16-bit DosSemSet and DosSemClear functions. If DosSemSet is passed a 32-bit semaphore handle, it will act exactly as if DosResetEventSem was called with the same handle. If DosSemClear is passed a 32-bit semaphore handle, it will act exactly as if DosPostEventSem was called with the same handle.

- Dos16SemClear will do a Dos32PostEventSem
- Dos16SemSet will do a Dos32ResetSem

## Using Semaphores for Mutual Exclusion

| Version | Functions |
|---|---|
| 32-bit | DosCreateMutexSem, DosOpenMutexSem, DosCloseMutexSem, DosRequestMutexSem, DosReleaseMutexSem, DosQueryMutexSem |

Mutual exclusion (mutex) semaphores protect critical sections of code from being executed by more than one thread at a time. The threads containing these critical sections of code can belong to the same process (a private semaphore), or to separate processes (a shared semaphore). Each process must call DosRequestMutexSem before beginning the critical section. If this function does not return an error, the caller has *ownership* of the semaphore and can proceed to

execute the critical section. When execution of the section is complete, the same thread must call DosReleaseMutexSem to give up ownership. The system will queue requests and transfer ownership to the next waiting thread when the semaphore is released. The semaphore also can be set when it is created, making the operation atomic and therefore completely reliable; with this approach, no other thread will run until the creating thread releases ownership.

The examples in Figure 4-11 and Figure 4-12 on page 4-24 demonstrate the use of mutex semaphores. Two processes that write to a single, shared region must coordinate their activities so that both are not writing at the same time (to do so would damage the data). By convention, process 1 creates the semaphore, and process 2 opens it.

```
Process 1:

/* Allocate shared memory for the semaphore structure. */

SEL selShared;
PDOSFSRSEM pdosfsrs;

DosAllocShrSeg(sizeof(DOSFSRSEM), "\\sharemem\\myshrseg",
    &selShared);
pdosfsrs = MAKEP(selShared, 0);
pdosfsrs->cb = sizeof(DOSFSRSEM);
pdosfsrs->tid = pdosfsrs->pid = 0;
pdosfsrs->client = pdosfsrs->cUsage = 0;
pdosfsrs->sem = 0L;

/* Execute other code. */
    .
    .
/* Critical section begins here. Use this function to block until
   semaphore is available. */

DosFSRamSemRequest(pdosfsrs, SEM_INDEFINITE_WAIT);

/* Read or write the shared data. */
    .
/* Done with critical code. */

DosFSRamSemClear(pdosfsrs);

/* Execute rest of program. At termination, free shared memory. */

DosFreeSeg(selShared);
```

*Figure 4-11 (Part 1 of 2). Using Semaphores for Mutual Exclusion—16-Bit Version*

```
Process 2:

/* Allocate shared memory for the semaphore structure. */

SEL selShared;
PDOSFSRSEM pdosfsrs;

DosGetShrSeg("\\sharemem\\myshrseg", &selShared);
pdosfsrs = MAKEP(selShared, 0);

/* Execute other code. */
   .
/* Critical section begins here. */

DosFSRamSemRequest(pdosfsrs, SEM_INDEFINITE_WAIT);

/* Read or write the shared data. */
   .
/* Done with critical code. */

DosFSRamSemClear(pdosfsrs);
/* Execute other code. */
.
/* When finished  executing critical code, free shared memory. */

DosFreeSeg(selShared);
```

*Figure 4-11 (Part 2 of 2). Using Semaphores for Mutual Exclusion—16-Bit Version*

```
Process 1:

HMTX hmtx = 0;    /* handle to event semaphore */

DosCreateMutexSem ("\\sem32\\mymutex", &hmtx, 0, FALSE);

/* Execute other code. */
   .
/* Critical section begins here. */

/* Block until available          */

DosRequestMutexSem(hmtx, SEM_INDEFINITE_WAIT);

/* Read or write the shared data. */
   .
/* Done with critical code. */

DosReleaseMutexSem(hmtx);

/* Execute rest of program. At termination, close semaphore. */

DosCloseMutexSem(hmtx);

Process 2:
HMTX hmtx = 0;    /* handle to event semaphore */

DosOpenMutexSem("\\sem32\\mymutex", &hmtx);

/* Execute other code. */
   .
/* Critical section begins here. */

/* Block until available          */

DosRequestMutexSem(hmtx, SEM_INDEFINITE_WAIT);

/* Read or write the shared data. */
   .
/* Done with critical code. */

DosReleaseMutexSem(hmtx);

/* Execute other code. */
   .
/* When finished executing critical code, close the semaphore. */

DosCloseMutexSem(hmtx);
```

*Figure 4-12. Using Semaphores for Mutual Exclusion—32-Bit Version*

## Using Semaphores for Multiple Waiting

| Version | Functions |
|---|---|
| 32-bit | DosCreateMuxWaitSem, DosOpenMuxWaitSem, DosCloseMuxWaitSem, DosWaitMuxWaitSem, DosAddMuxWaitSem, DosDeleteMuxWaitSem, DosQueryMuxWaitSem |

The multiple-wait (muxwait) functions enable an application to wait on more than one semaphore at a time. A typical use would be in a process that requires several resources in order to perform its work, and that no changes occur to those resources while they are in its possession. This is a multiple-wait on mutex semaphores. A group of event semaphores also can be waited on to enable a thread or process to be activated whenever an particular event occurs (for instance, a thread monitoring three sources of input might remain inactive until one of the sources receives input). The muxwait functions allow waiting for all of the indicated semaphores as well. Note that mutex and event semaphores cannot be mixed in a muxwait semaphore. Also, if any of the individual semaphores are private to a process, the muxwait semaphore also must be private to that process.

The example in Figure 4-13 shows a thread requesting simultaneous ownership of several resources. This example uses unnamed, private semaphores; they also could be shared or named-shared semaphores.

```
HMUX hmux;    /* handle to multiple-wait semaphore                          */
ULONG ulUser; /* only (WAIT_ANY) or last (WAIT_ALL) semaphore released */
SEMRECORD apsr[5]; /* array of mutex semaphores created elsewhere
                    * and copied to aSemRec
                    */

DosCreateMuxWaitSem((PSZ) NULL, &hmux, 3, apsr, DCMW_WAIT_ALL);
DosWaitMuxWaitSem(hmux, SEM_INDEFINITE_WAIT, &ulUser);

/*
 * When this call returns, the other threads using the resources have
 * all released their mutex semaphores; this thread can now make
 * simultaneous use of all three, knowing that no other thread is
 * accessing the data.
 */
```

*Figure 4-13 (Part 1 of 2). Using Semaphores for Multiple Waiting*

```
/* Critical code is executed. */
    .

/* Mutex semaphores are released when critical section is complete; the
 * use of a muxwait semaphore does not free the program from releasing
 * the individual mutex semaphores it represents when the critical section
 * has passed. */

/*  Other code is executed. */
    .

/* Finished with the semaphore for good. */
DosCloseMuxWaitSem(hmux);

The following code fragment shows a thread waiting for multiple sources of
input. The thread will wait for input from any one of three sources.

HMUX hmux;    /* handle to multiple-wait semaphore              */
ULONG ulUser; /* only (WAIT_ANY) or last (WAIT_ALL) semaphore   */
              /* that signaled*/

SEMRECORD apsr[5]; /* array of event semaphores created         */
                   /* elsewhere and copied to apsr              */


DosCreateMuxWaitSem((PSZ) NULL, &hmux, 3, apsr, DCMW_WAIT_ANY);

/* When this call returns, at least one of the input sources is ready. */

DosWaitMuxWaitSem(hmux, SEM_INDEFINITE_WAIT, &ulUser);

/* Thread examines ulUser to determine which input source caused the
 * wake up. (Alternatively, each event semaphore in the group can be
 * polled with the DosQueryEventSem function using a time-out of 0.)*/

/* Thread now processes the input from the ready source. */
    .

/* Finished with the semaphore. */

DosCloseMuxWaitSem(hmux);
```

*Figure 4-13 (Part 2 of 2). Using Semaphores for Multiple Waiting*

## Unnamed Pipes

The 16-bit function, DosMakePipe, and its corresponding 32-bit function, DosCreatePipe, perform the same action. In the 16-bit version of the operating system, it was suggested that the maximum size of the data passed through the pipe should be 64KB. This was based on the segment size maximum and the performance overhead of using a larger data size. This size restriction is removed in the 32-bit version of the operating system.

## Named Pipes

Named-pipe functions, which are listed in Table 4-5, perform the same actions. Some of the 32-bit names are different. Please note that there are no equivalent 32-bit functions for DosRawReadNmPipe and DosRawWriteNmPipe.

*Table 4-5. 16-Bit to 32-Bit Named-Pipe Functions*

| 16-Bit Name | 32-Bit Name |
|---|---|
| DosCallNmPipe | DosCallNPipe |
| DosConnectNmPipe | DosConnectNPipe |
| DosDisConnectNmPipe | DosDisConnectNPipe |
| DosMakeNmPipe | DosCreateNPipe |
| DosPeekNmPipe | DosPeekNPipe |
| DosQNmPHandState | DosQueryNPHState |
| DosQNmPipeInfo | DosQueryNPipeInfo |
| DosQNmPipeSemState | DosQueryNPipeSemState |
| DosRawReadNmPipe | N/A |
| DosRawWriteNmPipe | N/A |
| DosSetNmPHandInfo | DosSetNPHState |
| DosSetNmPipeSem | DosSetNPipeSem |
| DosTransactNmPipe | DosTransactNPipe |
| DosWaitNmPipe | DosWaitNPipe |

## Queues

The queue functions perform the same actions in both the 16-bit and 32-bit versions of the operating system. However, be aware of the following:

- Although both the 16-bit and 32-bit versions of the functions are functionally similar, they do not produce compatible system resources; that is, you cannot create a 16-bit queue and write to it by using the 32-bit DosWriteQueue function This applies between processes as well.
- Memory allocation in the 32-bit version of the operating system must be made with the 32-bit linear functions. DosAllocSharedMem must be used to allocate space for messages placed in the queue, rather than the segmented equivalent.
- If a NoWait of 1 is used when reading (removing data) or peeking (determining the data contents without removing the data) a queue, the associated semaphore should be an event semaphore. A NoWait of 1 tells the calling function to return if the queue has not data, rather than waiting for the queue to receive data before returning. In addition, if threads in other processes will write to the queue, the semaphore should be shared (either named or unnamed) and must be opened before the writing threads use DosWriteQueue. Typically, you will use this semaphore as part of a muxwait semaphore to be able to respond to any of several input sources (if the thread is waiting for something to appear in the queue, it should use a NoWait of 0).

Table 4-6 lists the queue functions.

*Table 4-6. 16-Bit to 32-Bit Queue Functions*

| 16-Bit Name | 32-Bit Name |
|---|---|
| DosCreateQueue | DosCreateQueue |
| DosOpenQueue | DosOpenQueue |
| DosCloseQueue | DosCloseQueue |
| DosPeekQueue | DosPeekQueue |
| DosPurgeQueue | DosPurgeQueue |
| DosQueryQueue | DosQueryQueue |
| DosReadQueue | DosReadQueue |
| DosWriteQueue | DosWriteQueue |

## Timers

The timer functions provide time and date access, suspend threads for a given time interval, and run asynchronous and interval timers. Changes to these functions are minor, resulting primarily from changes to the semaphore functions. Timer functions are listed in Table 4-7.

*Table 4-7. 16-Bit to 32-Bit Timer Functions*

| 16-Bit Name | 32-Bit Name |
|---|---|
| DosGetDateTime | DosGetDateTime |
| DosSetDateTime | DosSetDateTime |
| DosSleep | DosSleep |
| DosTimerAsync | DosAsyncTimer |
| DosTimerStart | DosStartTimer |
| DosTimerStop | DosStopTimer |

**16-bit** DosTimerAsync

**32-bit** DosAsyncTimer

These functions perform the same action. In the 32-bit version of the operating system, an event semaphore must be reset before its handle is passed to a timer function. When the timer expires, it will post the event semaphore. If the timer is to be used again, the semaphore must be reset first. The semaphore can be shared or private.

## Dynamic Linking

The 32-bit functions operate in a way that is consistent with the 16-bit versions. The 16-bit functions, DosGetMachineMode, DosGetVersion, and DosGetEnv have no 32-bit equivalents; these functions are unnecessary, because the 32-bit version of the operating system has no family application programming interface (FAPI). The environment information is in the thread information block, and the version numbers are accessible through DosQuerySysInfo. Table 4-8 on page 4-29 lists the 16-bit and 32-bit dynamic-linking functions.

*Table 4-8. 16-Bit to 32-Bit Dynamic-Linking Functions*

| 16-Bit Name | 32-Bit Name |
|---|---|
| DosGetResource2 | DosGetResource |
| DosFreeModule | DosFreeModule |
| DosFreeResource | DosFreeResource |
| DosLoadModule | DosLoadModule |
| DosGetProcAddr | DosQueryProcAddr |
| N/A | DosQueryProcType |
| DosGetModHandle | DosQueryModuleHandle |
| DosGetModName | DosQueryModuleName |
| DosQAppType | DosQueryAppType |
| N/A | DosQueryResourceSize |
| DosGetMachineMode | N/A |
| BadDynLink | N/A |
| DosGetVersion | N/A |
| DosGetEnv | N/A |

**16-bit** DosGetResource2, DosFreeResource

**32-bit** DosGetResource, DosFreeResource

The 32-bit version of the operating system uses resources in the same way as the 16-bit version of the system. The paged version of DosGetResource will return a pointer to the read-only memory that represents the resource, just as DosGetResource2 did. In the first 16-bit version of the system, resources were stored in individual segments that could be freed with DosFreeSeg. In the second 16-bit version of the system, resources are allocated with DosGetResource2, which will pack several resources into one segment, and are freed by using DosFreeResource. Under the 32-bit version of the system, resources are allocated with DosGetResource, which returns a 32-bit pointer to the resource. This pointer can be freed with DosFreeResource.

Note that the system will pack multiple resources into the same pages of memory and mark the access for those pages as PAG_READ | PAG_COMMIT.

The example in Figure 4-14 on page 4-30 shows the loading of a resource by a program, a sample use of the resource data, and the eventual discarding of the resource.

```
16-bit:

PSZ pszString;

DosGetResource2((HMODULE) NULL, RT_RCDATA, 1, &pszString);
ScreenPrint(hwnd, pszString);

/* ... */

DosFreeResource(pszString);

32-bit:

PSZ pszString;

DosGetResource((HMODULE) NULL, RT_RCDATA, 1, &pszString);
ScreenPrint(hwnd, pszString);

/* ... */

DosFreeResource(pszString);
```

*Figure 4-14. Using Resources*

## Device I/O

The DosBeep, DosDevConfig, and DosPhysicalDisk functions have remained basically the same. The DosCLIAccess and DosPortAccess functions are no longer valid in the 32-bit Version of the operating system. The DosDevIOCtl and DosDevIOCtl2 functions have been combined into the DosDevIOCtl function. Table 4-9 lists device I/O functions.

*Table 4-9. 16-Bit to 32-Bit Device I/O Functions*

| 16-Bit Name | 32-Bit Name |
|---|---|
| DosBeep | DosBeep |
| DosCLIAccess | N/A |
| DosPortAccess | N/A |
| DosDevConfig | DosDevConfig |
| DosPhysicalDisk | DosPhysicalDisk |
| DosDevIOCtl, DosDevIOCtl2 | DosDevIOCtl |

## File Systems

File-system functions were significantly improved in the 16-bit version of the system, with the introduction of installable file systems, the High Performance File System, and extended attributes. In the 32-bit version of the system, file-system functions are basically the same. One new function has been added: DosCancelLockRequest. This function allows a process to cancel the lock range request for an outstanding DosSetFileLocks function. Table 4-10 on page 4-31 lists the file-systems functions.

*Table 4-10 (Page 1 of 2). 16-Bit to 32-Bit File-System Functions*

| 16-Bit Name | 32-Bit Name |
|---|---|
| DosBufReset | DosResetBuffer |
| DosChDir | DosSetCurrentDir |
| DosChgFilePtr | DosSetFilePtr |
| DosClose | DosClose |
| DosCopy | DosCopy |
| DosDelete | DosDelete |
| DosDupHandle | DosDupHandle |
| DosEditName | DosEditName |
| DosEnumAttribute | DosEnumAttribute |
| DosFileIO | N/A |
| DosFileLocks | DosSetFileLocks |
| DosFindClose | DosFindClose |
| DosFindFirst | DosFindFirst |
| DosFindNext | DosFindNext |
| DosFindNotifyClose | N/A |
| DosFindNotifyFirst | N/A |
| DosFindNotifyNext | N/A |
| DosFSAttach | DosFSAttach |
| DosFSCtl | DosFSCtl |
| DosMkDir | DosCreateDir |
| DosMove | DosMove |
| DosNewSize | DosSetFileSize |
| DosOpen | DosOpen |
| DosQCurDir | DosQueryCurrentDir |
| DosQCurDisk | DosQueryCurrentDisk |
| DosQFHandState | DosQueryFHState |
| DosQFileInfo | DosQueryFileInfo |
| DosQFileMode | N/A |
| DosQFSAttach | DosQueryFSAttach |
| DosQFSInfo | DosQueryFSInfo |
| DosQHandType | DosQueryHType |
| DosQPathInfo | DosQueryPathInfo |
| DosQSysInfo | DosQuerySysInfo |
| DosQVerify | DosQueryVerify |
| DosRead | DosRead |
| DosReadAsync | N/A |
| DosRmDir | DosDeleteDir |
| DosScanEnv | DosScanEnv |
| DosSearchPath | DosSearchPath |
| DosSelectDisk | DosSetDefaultDisk |

*Table 4-10 (Page 2 of 2). 16-Bit to 32-Bit File-System Functions*

| 16-Bit Name | 32-Bit Name |
|---|---|
| DosSetFHandState | DosSetFHState |
| DosSetFileInfo | DosSetFileInfo |
| DosSetFileMode | N/A |
| DosSetFsInfo | DosSetFSInfo |
| DosSetMaxFH | DosSetMaxFH, DosSetRelMaxFH |
| DosSetPathInfo | DosSetPathInfo |
| DosSetVerify | DosSetVerify |
| DosShutDown | DosShutdown |
| DosWrite | DosWrite |
| DosWriteAsync | N/A |
| N/A | DosCancelLockRequest |

## Searching Directories

| Version | Functions |
|---|---|
| **16-bit** | DosFindFirst, DosFindNext, DosFindClose |
| **32-bit** | DosFindFirst, DosFindNext, DosFindClose |

In the 16-bit version of the system, the DosFindFirst and DosFindNext functions will return all "normal" files matching a search string. (A "normal" file is one with no attribute bits set.) However, files marked as *archived*, *read-only*, *system*, or *hidden*, or subdirectories that are to be included in the search, must have, at least, their corresponding attribute bits marked to be returned.

The same logic applies for the 32-bit version of the system, but a new flag has been added to exclude normal files from the search. When this flag is used, only files matching the set of attribute bits selected for the search are returned. This can improve local and network file efficiency. Figure 4-15 shows an example of code used in the 32-bit version of the system to find subdirectories.

```
ULONG ulSearchCount = 256;
HDIR hdir = HDIR_CREATE;
BYTE abFileInfo[4096];
PFILEFINDBUF pfindbuf;

pfindbuf = (PFILEFINDBUF) abFileInfo;

DosFindFirst("*",
             &hdir,
             0x80 | FILE_DIRECTORY | FILE_HIDDEN | FILE_SYSTEM,
             pfindbuf,
             sizeof(abFileInfo),
             &ulSearchCount,
             0);
```

*Figure 4-15. 32-Bit Code for Finding Directories*

In addition, the 32-bit version of the system has a new information level available to applications: level 4, "Return All EAs." (EAs refer to extended attributes.) ResultBuf is undefined on input but on output, it will contain a packed set of records that consist of the attributes record described in level 1, followed by an FEAList (full extended attribute list) structure that contains all the extended-attribute information for the file. The length field of this FEA list is valid, giving the size of the FEA-list buffer.

### Querying File Mode

| Version | Functions |
|---|---|
| 16-bit | DosQFileMode |
| 32-bit | DosQueryFileInfo |

The attribute information returned by DosQFileMode in the 16-bit version of the system should be obtained by using information level 1. Use level 2 with DosQueryFileInfo in the 32-bit version of the system.

### Querying System Information

| Version | Functions |
|---|---|
| 16-bit | DosQSysInfo |
| 32-bit | DosQuerySysInfo |

In the 32-bit version of the system, a new system constant describing the maximum size of a path-name component has been added. Applications that hard-coded this value to 255 with the 16-bit version of the system can now request this information from the system. The system can now change the value without causing the application to change.

### Reading Asynchronously

| Version | Functions |
|---|---|
| 16-bit | DosReadAsync |
| 32-bit | No equivalent |

This function does not exist in the 32-bit version of the system. An application that cannot afford to be blocked during a read operation should create a separate thread for the reading process and post an event semaphore when the reading is complete. For example, an application may be holding serialization over a shared resource. If the application is blocked because of a synchronous read request, all other requestors of the serialization will also be blocked until the I/O is completed. This could lead to a potentially serious performance degradation.

### Setting the File Mode

| Version | Functions |
|---|---|
| 16-bit | DosSetFileMode |
| 32-bit | No equivalent |

In the 32-bit version of the system, use DosSetFileInfo with information-level 1 to change the file mode.

### Setting Available Number of File Handles

| Version | Functions |
|---|---|
| 16-bit | DosSetMaxFH |
| 32-bit | DosSetMaxFH, DosSetRelMaxFH |

The maximum number of file handles available to a process (and, by default, its child processes) can be set with DosSetMaxFH. In the 32-bit version of the system, processes can adjust the maximum number in a relative fashion (by incrementing or decrementing the current value regardless of what the current value is), allowing a more natural interface for allocating and deallocating the resource. Also, by specifying a relative change of zero handles, a process can query the current maximum.

### Writing Asynchronously

| Version | Functions |
|---|---|
| 16-bit | DosWriteAsync |
| 32-bit | No equivalent |

This function does not exit in the 32-bit version of the system. An application that cannot afford to be blocked during a write operation should create a separate thread for the writing process and post an event semaphore when the writing is complete.

## Message Retrieval

Table 4-11 lists the 16-bit and 32-bit message-retrieval functions.

*Table 4-11. 16-Bit to 32-Bit Message-Retrieval Functions*

| 16-Bit Name | 32-Bit Name |
|---|---|
| DosGetMessage | DosGetMessage |
| DosInsMessage | DosInsertMessage |
| DosPutMessage | DosPutMessage |
| N/A | DosQueryMessageCp |

## Code-Page Management

Table 4-12 lists the 16-bit and 32-bit code-page functions.

*Table 4-12. 16-Bit to 32-Bit Code-Page Management Functions*

| 16-Bit Name | 32-Bit Name |
|---|---|
| DosSetCp | N/A |
| DosSetProcCp | DosSetProcessCp |
| DosGetCp | DosQueryCp |
| DosGetCtryInfo | DosQueryCtryInfo |
| DosCaseMap | DosMapCase |
| DosGetDBCSEv | DosQueryDBCSEnv |
| DosGetCollate | DosQueryCollate |

## Session Management

Sessions are groups of processes that together "own" a *virtual* screen. A virtual screen refers to a physical hardware monitor that is emulated to all types of applications: DOS, VIO, AVIO, and PM. In general, only full-screen virtual input output (VIO) applications use session management APIs; all PM applications run inside the PM session and do not normally call any session functions.

Table 4-13 lists the 16-bit and 32-bit session-management functions.

*Table 4-13. 16-Bit to 32-Bit Session-Management Functions*

| 16-Bit Name | 32-Bit Name |
|---|---|
| DosStartSession | DosStartSession |
| DosSetSession | DosSetSession |
| DosSelectSession | DosSelectSession |
| DosStopSession | DosStopSession |
| DosSMRegisterDD | N/A |

## Error Management

A process can use error-management functions to disable hard-error and exception notification to the user, preferring instead to deal with the errors. In the event of an error, an application then can ignore the error, recover from the error, and provide a more descriptive indication of error status or a more user-friendly error message. In this way, the application can have a more detailed understanding of the error than the system and therefore can "fine tune" the correct action in response to the error. In the 32-bit version of the system, the information is encoded as binary flags (bit 0 is 1 to enable hard-error pop-up messages; bit 1 is 1 to enable exception pop-up messages). In both versions, the default is to enable exception and hard-error pop-up messages. Table 4-14 lists the 16-bit and 32-bit error-management functions.

*Table 4-14. 16-bit to 32-bit Error Management Functions*

| 16-Bit Name | 32-Bit Name |
|---|---|
| DosErrClass | DosErrClass |
| DosError | DosError |

## Signals

Signals have been implemented into the exception-management architecture. The signals 'CTRL+C', 'CTRL+BREAK', and 'KILLPROCESS' have been defined as signal exceptions.

The 16-bit signal functions are DosHoldSignal, DosSetSigHandler, and DosSendSignal. There are no equivalent 32-bit functions.

## Exception Management

In the 16-bit version of the operating system, signals are dispatched from a number of sources and are used to interrupt executing processes. These signals include "CTRL+C," "CTRL+BREAK," and "KILLPROCESS."

In the 32-bit version of the operating system, signal handling has been merged with exception handling to provide a general, portable, mechanism for handling all such events. Four new system functions are defined for exception handling on a per-thread basis:

- DosRaiseException()
- DosSetExceptionHandler()
- DosUnsetExceptionHandler()
- DosUnwindException()

Applications can register their own routines (by using DosSetExceptionHandler) to handle specific types of exceptions, including general protection exceptions that cannot be trapped by applications under previous versions of the operating system. Instead of requiring that exception handlers be written in assembly language (as is the case with the 16-bit version of the operating system), they can now be written in a high-level language, such as C.

In the 32-bit version of the operating system, general protection exceptions can be handled within the application. This allows an application to recover or at least to terminate in an orderly manner.

The DosSetSignalExceptionFocus function is used by 32-bit applications to inform the operating system that it is ready to receive signals. However, these signals are dispatched and treated as exceptions. This generalized approach enables exception handlers to be chained, nested, or both. Exceptions can then be handled by the first exception handler in the chain, or passed to subsequent handlers. The fact that these exception handlers can be written in a high-level language reduces dependencies on the 80386 architecture, and eases the porting of applications to other operating systems and hardware platforms.

Table 4-15 lists the 16-bit and 32-bit exception-management functions. There are few corresponding 16-bit exception-management functions.

*Table 4-15. 16-Bit to 32-Bit Exception-Management Functions*

| 16-Bit Name | 32-Bit Name |
|---|---|
| DosSetVec | N/A |
| N/A | DosSetExceptionHandler |
| N/A | DosUnsetExceptionHandler |
| N/A | DosRaiseException |
| N/A | DosUnwindException |
| N/A | DosSendSignalException |
| N/A | DosSetSignalExceptionFocus |
| N/A | DosAcknowledgeSignalException |
| N/A | DosEnterMustComplete |
| N/A | DosExitMustComplete |

## VDD Services

The 32-bit version of the system allows protect-mode processes to request services directly from a virtual device driver (VDD). The new functions shown in Table 4-16 support this capability.

*Table 4-16. 32-Bit VDD Services*

| 16-Bit Name | 32-Bit Name |
|---|---|
| N/A | DosOpenVDD |
| N/A | DosRequestVDD |
| N/A | DosCloseVDD |

## Support for 16-Bit Subsystems

OS/2 2.0 provides support for the following 16-bit subsystems:

Vio (video subsystem)
Kbd (keyboard subsystem)
Mou (mouse subsystem)
Mon (device-monitor subsystem)

The functions of these subsystems are described in the *OS/2 Version 1.3 Programming Tools and Information.*

## Changes to Presentation Manager Services

Changes to PM services can be categorized as new functions and as obsolete 16-bit functions. Some 16-bit PM functions do not exist in the 32-bit PM function set, but are are still available to 16-bit applications running under the 32-bit version of the system. New functions are for 32-bit applications only.

The 16-bit functions that are not included in the 32-bit PM function set are as follows:

- Heap Management
  - WinCreateHeap
  - WinDestroyHeap
  - WinAllocMem
  - WinAvailMem
  - WinFreeMem
  - WinLockHeap
  - WinReallocMem

    **Note:** These functions are replaced by 32-bit memory-management functions.

- Installed Program List
  - PrfCreateGroup
  - PrfDestroyGroup
  - PrfAddProgram
  - PrfRemoveProgram
  - PrfChangeProgram
  - PrfQueryProgramCategory
  - PrfQueryProgramHandle
  - PrfQueryProgramTitles
  - PrfQueryDefinition
  - WinAddProgram

- WinCreateGroup
- WinInstStartApp
- WinQueryDefinition
- WinQueryProgramTitles
- WinTerminateApp

  **Note:** These functions are replaced by new workplace functions.

- Initialization File
  - WinQueryProfileData
  - WinQueryProfileInt
  - WinQueryProfileSize
  - WinQueryProfileString
  - WinWriteProfileData
  - WinWriteProfileString

  **Note:** These functions have been replaced by 32-bit profile functions.

- Window Locking
  - WinLockWindow
  - WinLockWindowUpdate
  - WinQueryWindowLockCount

  **Note:** These functions have no corresponding 32-bit versions.

- Window Management
  - WinRegisterWinDestroy

  **Note:** This function has been replaced with a hook

New 32-bit PM functions are available for printing, application integration and object management, customizing help information, 32-bit migration, standard dialogs, and dragging and dropping objects. In addition, a new set of controls and hooks, as well as helper macros, have been defined.

## Printing

Applications can control print spooling locally and on a network with Spooler functions. Spooler functions enable applications to manipulate printer objects (print destinations or printers, print jobs, and print queues). Applications can add, delete, query, and modify any printer object. Applications also can interrupt or continue jobs or groups of jobs, and can purge all jobs from a queue.

Spooler functions remain the same except for name changes. Table 4-17 on page 4-39 lists the functions.

*Table 4-17. Version 2.0 Print Functions*

| 16-Bit Name | 32-Bit Name |
|---|---|
| DosPrintDestControl | SplControlDevice |
| DosPrintDestAdd | SplCreateDevice |
| DosPrintDestDel | SplDeleteDevice |
| DosPrintDestEnum | SplEnumDevice |
| DosPrintDestGetInfo | SplQueryDevice |
| DosPrintDestSetInfo | SplSetDevice |
| DosPrintJobDel | SplDeleteJob |
| DosPrintJobEnum | SplEnumJob |
| DosPrintJobPause | SplHoldJob |
| DosPrintJobGetId | SplQueryJobId |
| DosPrintJobGetInfo | SplQueryJob |
| DosPrintJobContinue | SplReleaseJob |
| DosPrintJobSetInfo | SplSetJobInfo |
| DosPrintQAdd | SplCreateQueue |
| DosPrintQDel | SplDeleteQueue |
| DosPrintQEnum | SplEnumQueue |
| DosPrintQPause | SplHoldQueue |
| DosPrintQPurge | SplPurgeQueue |
| DosPrintQGetInfo | SplQueryQueue |
| DosPrintQContinue | SplReleaseQueue |
| DosPrintQSetInfo | SplSetQueue |
| DosPrintDriverEnum | SplEnumDriver |
| DosPrintPortEnum | SplEnumPort |
| DosPrintQProcessorEnum | SplEnumQueueProcessor |

## Workplace

Applications can define objects and what user actions the applications will perform on those objects. PM provides default handling for objects. Table 4-18 on page 4-40 lists the new object-management functions.

*Table 4-18. OS/2 2.0 Workplace Functions*

| Name | Function Description |
|---|---|
| WinCreateObject | Create an object class |
| WinDeregisterObjectClass | Remove a workplace object class |
| WinDestroyObject | Delete a workplace object |
| WinEnumObjectClasses | Return a list of all workplace object classes that have been registered |
| WinFreeFileIcon | Free the pointer to an icon allocated by WinLoadFileIcon |
| WinLoadFileIcon | Return a pointer to an icon which is associated with the file specified |
| WinRegisterObjectClass | Register a workplace object class |
| WinReplaceObjectClass | Replace a registered class with another registered class |
| WinRestoreWindowPos | Restore the size and position of the window specified |
| WinSetFileIcon | Set the icon for the specified file |
| WinSetObjectData | Set data on a workplace object |
| WinShutdownSystem | Close down the system |
| WinStoreWindowPos | Save the current size and position of the specified window |

## Customizing Help Information

The Information Presentation Facility (IPF) manages online, context-sensitive help information. It displays both text and graphics and provides hypertext links (selectable words or phrases that connect related information) within and between information units. Enhancements to IPF for the 32-bit version of the system include new tags that provide:

- Multiple-viewport support.
- Device-independent graphics support through *metafiles*. A metafile is a generic name for the definition of the contents of a graphics file.
- Horizontal scrolling in help windows.
- Multiple text fonts and sizes.
- Hypertext links across files.

In addition, authors can customize help information with:

- Application-controlled viewports: for animation, full-motion video, simulations, and so on.
- Dynamic data formatting (DDF): for displaying dynamic help information.

Customized help information is created from a tagged source file that defines each help window (including help, next, previous, search, print, and other buttons), and from application code that is executed by IPF when it processes specific tags in the tagged source file. Table Table 4-19 on page 4-41 summarizes the new IPF tags.

*Table 4-19. New IPF Tags*

| Name | Tag Description |
|---|---|
| :acviewport | Enables an application to dynamically control what is displayed in an IPF window |
| :br | Causes a break in a line of text |
| :ddf | Display dynamically formatted text in an application-controlled window |
| :docprof | Specifies the heading-level entries to be displayed in the Contents window |
| :font | Changes the font to the specified typeface, size, and code page. |
| :table | Formats information as a table |
| :title | Provides a name for the online document |

In addition, dynamic data formatting is accomplished through a new set of functions. Table Table 4-20 summarizes the new DDF functions.

*Table 4-20. Version 2.0 DDF Functions*

| Name | Function Description |
|---|---|
| DdfBeginList | Begin a definition list |
| DdfBitmap | Place a reference to a bit map in the DDF buffer |
| DdfEndList | End a definition list |
| DdfHyperText | Define a hypertext link to another panel |
| DdfInform | Define a hypertext inform link |
| DdfInitialize | Initialize the IPF internal structures for dynamic data formatting and returns a DDF handle. |
| DdfListItem | Insert a definition list entry in the DDF buffer |
| DdfMetafile | Place a reference to a metafile in the DDF buffer |
| DdfPara | Create a paragraph within the DDF buffer |
| DdfSetColor | Set the background and foreground colors of displayed text |
| DdfSetFont | Specify a text font in the DDF buffer |
| DdfSetFontStyle | Specify a text font style in the DDF buffer |
| DdfSetFormat | Turn formatting off or on |
| DdfSetTextAlign | Define whether left, center, or right text justification is to be used when the text formatting is off. |
| DdfText | Add text to the DDF buffer |

## 32-Bit Migration

In a mixed-model environment, 16-bit code interacts with 32-bit code, and 16-bit window procedures send messages to 32-bit window procedures. For system-defined messages, PM performs the necessary translation through a thunk layer. For user-defined messages, however, applications must create 16-bit to 32-bit thunk procedures to manage 16-bit window-message parameters, to convert 16-bit segmented pointer parameters to 32-bit flat pointers and, if necessary, to convert structures pointed to by parameters. Applications provide the thunking layer through new migration functions, which are listed in Table 4-21 on page 4-42.

| Table 4-21. Version 2.0 Migration Functions | |
|---|---|
| **Name** | **Description** |
| WinQueryWindowMode | Query the memory model associated with a window |
| WinSetWindowThunkProc | Set a thunk procedure for a window |
| WinQueryWindowThunkProc | Query a thunk procedure for a window |
| WinSetClassThunkProc | Associate a thunk procedure with a window class |
| WinQueryClassThunkProc | Query the pointer-conversion procedure associated with a window class |

## Standard Font-and File-Dialog Boxes

In OS/2 2.0, PM provides functions that enable applications to create standard font and file dialog boxes. Standard dialogs boxes across applications save application code and increase user productivity through a consistent interface. Standard dialogs boxes are extendable (applications can add controls) and customizable (applications can change button text, window title text).

File dialog functions request file names from users and perform file-name validation. Applications initialize fields and filter strings, and can specify modal or nonmodal dialog boxes and single or multiple-file selections. A file dialog can be implemented as either an *Open* or a *SaveAs* dialog. The following figures are examples of these two dialogs.

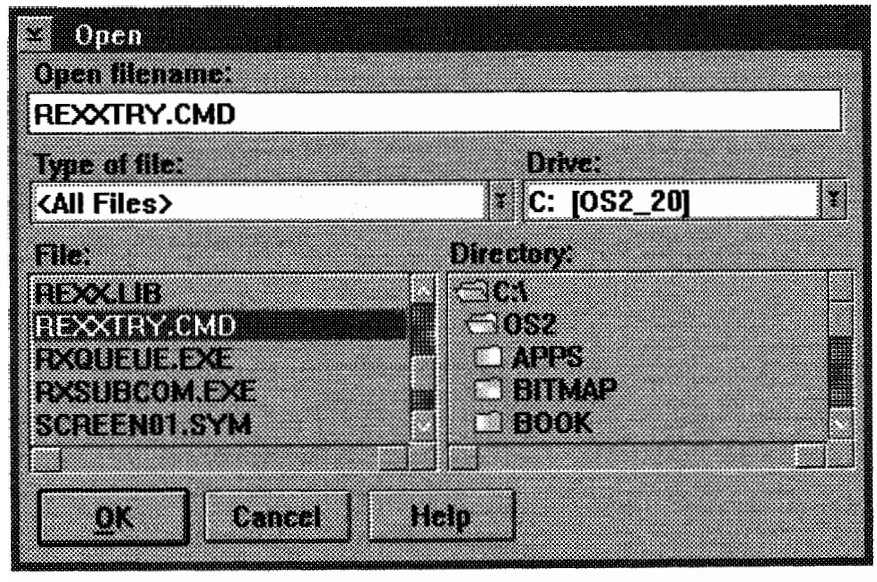



*Figure 4-16. Open Dialog*

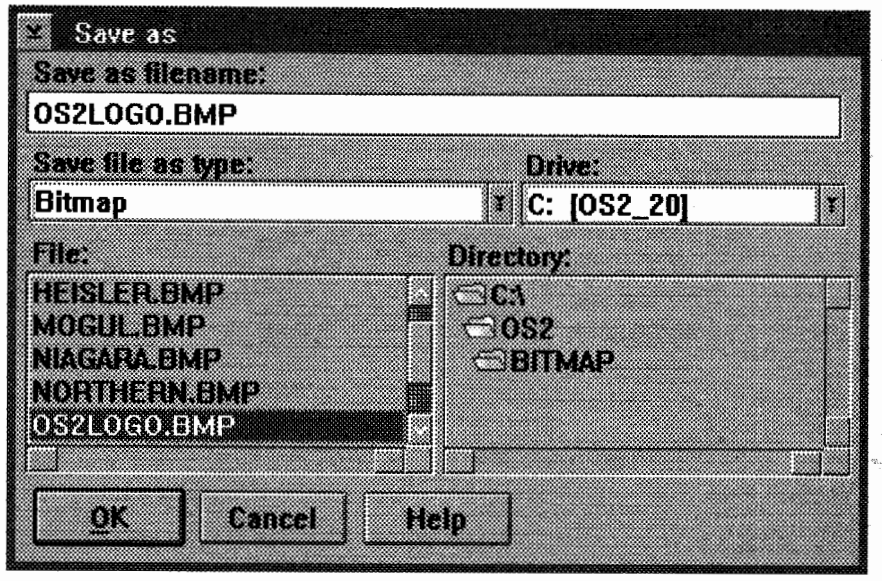



*Figure 4-17. SaveAs Dialog*

Font dialog boxes request font definitions from users, provide preview windows, and return font face names, point size, boldness, and other specifications. Applications can specify modal or nonmodal dialogs, color selection functions, and single or multiple font selection. Figure 4-18 shows a sample font dialog box.

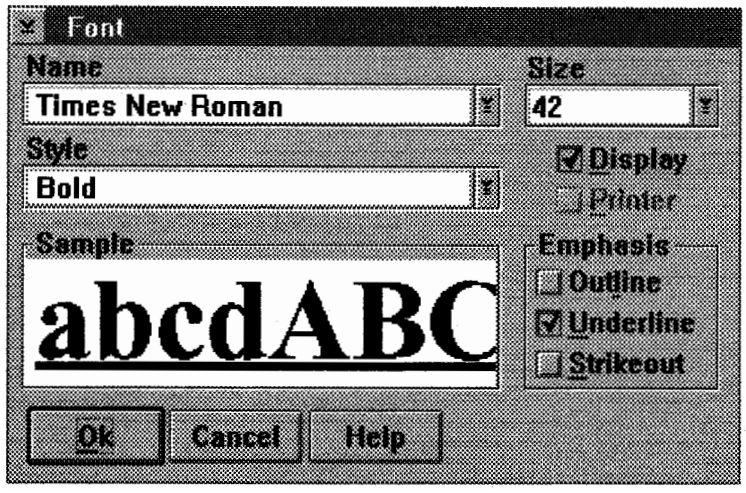



*Figure 4-18. Font Dialog*

Table 4-22 lists the new font-and file-dialog functions.

*Table 4-22. Font and File Dialog Functions*

| Name | Description |
|---|---|
| WinFileDlg | Create and display the file dialog box and return the user's selection or selections |
| WinFontDlg | Allow a user to select a font |

## Window Controls

The following new window controls have been defined for PM in OS/2 2.0: notebook, container, value set, and slider.

### Notebook

This window control organizes access to multiple groups of controls. The overall appearance of this control is a notebook. An application can dynamically insert or delete pages, specify colors for different notebook areas, and resize parts of the notebook. The content of each page is defined and managed by the application. Figure 4-19 shows an example of a notebook control window.

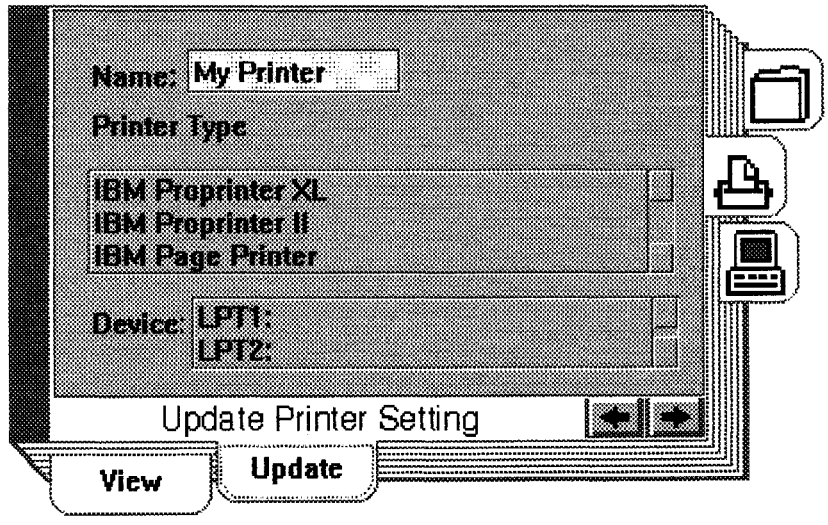



*Figure 4-19. Notebook Control Window*

Figure 4-20 lists the window class, styles, and messages associated with this new window control.

```
Window Class
    WC_NOTEBOOK

Window Styles
    Back-Page Intersection:
        BKS_BACKPAGESBR
        BKS_BACKPAGESBL
        BKS_BACKPAGESTR
        BKS_BACKPAGESTL
```

*Figure 4-20 (Part 1 of 2). Control Window Classes, Styles, and Messages*

```
    Major-Tab Side:
       BKS_MAJORTABRIGHT
       BKS_MAJORTABLEFT
       BKS_MAJORTABTOP
       BKS_MAJORTABBOTTOM
    Shape of Notebook Tabs:
       BKS_SQUARETABS
       BKS_ROUNDEDTABS
       BKS_POLYGONTABS
    Position of Status-Line Text:
       BKS_STATUSTEXTLEFT
       BKS_STATUSTEXTRIGHT
       BKS_STATUSTEXTCENTER

Window Messages

    Notebook Notification:
       BKN_NEWPAGESIZE
       BKN_PAGESELECTED
    Notebook Page:
       BKM_CALCPAGERECT
       BKM_DELETEPAGE
       BKM_INSERTPAGE
       BKM_INVALIDATETABS
       BKM_TURNTOPAGE
    Query Notebook:
       BKM_QUERYPAGECOUNT
       BKM_QUERYPAGEID
       BKM_QUERYPAGEULONG
       BKM_QUERYPAGEWINDOWHWND
       BKM_QUERYTABBITMAP
       BKM_QUERYTABTEXT
    Set Notebook:
       BKM_SETDIMENSIONS
       BKM_SETPAGEULONG
       BKM_SETPAGEWINDOWHWND
       BKM_SETSTATUSLINETEXT
       BKM_SETTABBITMAP
       BKM_SETTABTEXT
```

*Figure 4-20 (Part 2 of 2). Control Window Classes, Styles, and Messages*

## Container

A container control is a visual component whose specific purpose is to hold objects. A container can display objects in various formats and views. Each view generally presents different information about each object.

This window control displays and processes the user's selection of objects. Selection can be defined as autoselect, extended select, marquee select, multiple select, or single select. This control supports direct manipulation of objects, enabling users to drag an object from a container window and drop it on another object or container window. Figure 4-21 on page 4-46 shows an example of a container control.


*Figure 4-21. Container Control Window*

Figure 4-22 lists the window class, styles and messages associated with this new window control.

```
Window Class
    WC_CONTAINER

Window Styles
    Control Styles:
        CS_AUTOPOSITION
        CS_AUTOSELECTION
        CS_READONLY
        CS_VERIFYPOINTERS
    Selection Types:
        CS_SINGLESEL
        CS_EXTENDSEL
        CS_MULTIPLESEL

Window Messages
    Container Notification:
        CN_DRAGAFTER
        CN_DRAGLEAVE
```

*Figure 4-22 (Part 1 of 2). Container Control Window Classes, Styles, and Messages*

```
    CN_DRAGOVER
    CN_DROP
    CN_DROPHELP
    CN_EMPHASIS
    CN_ENDEDIT
    CN_ENTER
    CN_INITDRAG
    CN_KILLFOCUS
    CN_QUERYDELTA
    CN_REALLOCPSZ
    CN_SCROLL
    CN_SETFOCUS
Record:
    CM_ALLOCRECORD
    CM_FREERECORD
    CM_INSERTRECORD
    CM_INVALIDATERECORD
    CM_QUERYRECORD
    CM_REMOVERECORD
Details View Columns:
    CM_ALLOCDETAILFIELDINFO
    CM_FREEDETAILFIELDINFO
    CM_HORZSCROLLSPLITWINDOW
    CM_INSERTDETAILFIELDINFO
    CM_INVALIDATEDETAILFIELDINFO
    CM_QUERYDETAILFIELDINFO
    CM_REMOVEDETAILFIELDINFO
General Container:
    CM_ARRANGE
    CM_CLOSEEDIT
    CM_ERASERECORD
    CM_FILTER
    CM_OPENEDIT
    CM_PAINTBACKGROUND
    CM_QUERYCNRINFO
    CM_QUERYDRAGIMAGE
    CM_QUERYRECORDEMPHASIS
    CM_QUERYRECORDFROMRECT
    CM_QUERYRECORDRECT
    CM_QUERYVIEWPORTRECT
    CM_SCROLLWINDOW
    CM_SEARCHSTRING
    CM_SETCNRINFO
    CM_SETRECORDEMPHASIS
    CM_SORTRECORD
```

*Figure 4-22 (Part 2 of 2). Container Control Window Classes, Styles, and Messages*

## Value Set

A value-set control is a visual component that enables a user to select one choice from a group of mutually-exclusive choices. A value set can use graphic images (bit maps or icons), as well as colors, text, and numbers, to represent the items that a user can select. An application can specify different types of items, sizes, and orientations for its value sets. Figure 4-23 shows an example of a value-set control.

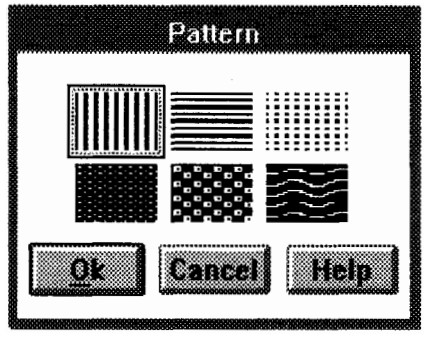



*Figure 4-23. Value-Set Control*

Figure 4-24 lists the window class, styles, and messages associated with this new window control.

```
Window Class

    WC_VALUESET

Window Styles

    VS_BITMAP
    VS_ICON
    VS_TEXT
    VS_RGB
    VS_COLORINDEX
    VS_BORDER
    VS_ITEMBORDER
    VS_RIGHTTOLEFT

Messages

    Value Set Notification:
        VN_DRAGLEAVE
        VN_DRAGOVER
        VN_DROP
        VN_DROPHELP
        VN_ENTER
        VN_INITDRAG
        VN_KILLFOCUS
        VN_SELECT
        VN_SETFOCUS
```

*Figure 4-24 (Part 1 of 2). Value-Set Control Window Classes, Styles, and Messages*

```
Query Value Set:
    VM_QUERYITEM
    VM_QUERYITEMATTR
    VM_QUERYMETRICS
    VM_QUERYSELECTEDITEM
Set Value Set:
    VM_SELECTITEM
    VM_SETITEM
    VM_SETITEMATTR
    VM_SETMETRICS
```

*Figure 4-24 (Part 2 of 2). Value-Set Control Window Classes, Styles, and Messages*

## Slider

A slider control enables a user to set, display, or modify a value by moving a slider arm. The appearance of a slider, and how the user uses it, is similar to a scroll bar. However, they are not interchangeable because each has a distinct purpose. The scroll bar makes visible information that is outside a window's client area; the slider is used to set, display, or modify information, whether it is in the client area or not. Applications can specify different scales, sizes, and orientations for its sliders. Figure 4-25 shows an example of a slider control.

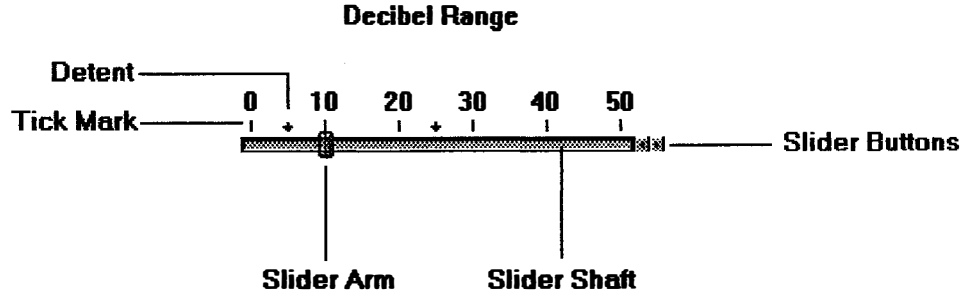



*Figure 4-25. Slider Control*

Figure 4-26 lists the window class, styles, and messages associated with this new window control.

```
Window Class

    WC_SLIDER

Window Styles

    Orientation:
        SLS_HORIZONTAL
        SLS_VERTICAL
```

*Figure 4-26 (Part 1 of 2). Slider Control Window Classes, Styles, and Messages*

```
    Position:
        SLS_CENTER
        SLS_BOTTOM
        SLS_TOP
        SLS_LEFT
        SLS_RIGHT
    Scale Location on Shaft:
        SLS_PRIMARYSCALE1
        SLS_PRIMARYSCALE2
    Slider-Arm's Home Position:
        SLS_HOMELEFT
        SLS_HOMERIGHT
        SLS_HOMEBOTTOM
        SLS_HOMETOP
    Slider-Button Locations:
        SLS_BUTTONSLEFT
        SLS_BUTTONSRIGHT
        SLS_BUTTONSBOTTOM
        SLS_BUTTONSTOP
    Miscellaneous:
        SLS_SNAPTOINCREMENT
        SLS_READONLY
        SLS_RIBBONSTRIP
        SLS_OWNERDRAW

Window Messages

    Slider Notification:
        SLN_CHANGE
        SLN_SLIDERTRACK
        SLN_KILLFOCUS
        SLN_SETFOCUS
    Query Slider:
        SLM_QUERYDETENTPOS
        SLM_QUERYSCALETEXT
        SLM_QUERYSLIDERINFO
        SLM_QUERYTICKPOS
        SLM_QUERYTICKSIZE
    Set Slider:
        SLM_SETSCALETEXT
        SLM_SETSLIDERINFO
        SLM_SETTICKSIZE
    Add or Remove Detent:
        SLM_ADDETENT
        SLM_REMOVEDETENT
```

*Figure 4-26 (Part 2 of 2). Slider Control Window Classes, Styles, and Messages*

## Pop-Up Menus

In OS/2 2.0, PM provides a function, WinPopupMenu, that enables an application to display a pop-up menu. A pop-up menu is a menu that, when requested, is displayed next to the object with which it is associated. It contains choices appropriate for a given object or set of objects in their current context.

## Desktop Background

In OS/2 2.0, PM provides functions that enables applications to fill the desktop background with a specified bit map. Bit maps can be centered, tiled, or scaled in the desktop background. The desktop background functions are WinSetDesktopBkgnd and WinQueryDesktopBkgnd. These functions will only work when the Workplace Shell is not active, and can be used by an application that replaces the shell.

## Hooks

In OS/2 2.0, PM provides new window-manager hook capabilities. Applications can apply message filtering when WinGetMsg, WinPeekMsg, or WinWaitMsg is called through the hook, CheckMsgFilterHook. This enables applications to examine new messages in the main event loop between the time they retrieve the message from the queue and the time they dispatch it, performing special processing as appropriate. Applications also can filter the window-destroy process with the hook, DestroyWindowHook. This hook is called after the WM_DESTROY message is processed but before the window handle becomes invalid. It replaces the WinRegisterWindowDestroy function.

## Paths, Regions, and Bit Maps

In OS/2 2.0, PM provides new functions for paths, regions, and bit maps:

- Paths can be converted into regions.
- More operations can be performed on regions.
- Areas can be easily filled.
- A bit map can be copied from storage to a bit map in a device context.

Table 4-23 lists these new functions.

*Table 4-23. Path, Region, and Bit Map Functions*

| Name | Function Description |
|---|---|
| GpiDrawBits | Draw a rectangle of bits |
| GpiFloodFill | Fill an area bounded by a given color, or while on a given color |
| GpiFrameRegion | Draw a frame inside a region using the current pattern attributes |
| GpiPathToRegion | Convert a path into a region |

## Fonts and Characters

In OS/2 2.0, PM provides new functions that give applications greater ability to inquire about logical fonts and face names, the ability to modify the default font, and extra character spacing for text justification. Table 4-24 on page 4-52 lists the new functions.

| Table 4-24. Font and Character Functions | |
|---|---|
| **Name** | **Function Description** |
| GpiLoadPublicFonts | Load one or more fonts from the specified resource file, to be available for all applications |
| GpiQueryCharBreakExtra | Return the current value of the character-break-extra attribute |
| GpiQueryCharExtra | Return the current value of the character-extra attribute |
| GpiQueryCharOutline | Return the drawing order of a character outline |
| GpiQueryFaceString | Generate a compound face name for a font |
| GpiQueryLogicalFont | Return the description of a logical font |
| GpiSetCharBreakExtra | Specify an extra increment to be used for spacing break characters in a string, for example, space characters |
| GpiSetCharExtra | Specify an extra increment to be used for spacing characters in a string |
| GpiUnloadPublicFonts | Unload one or more generally-available fonts |

## Polylines

| Table 4-25. Polyline Functions | |
|---|---|
| **Name** | **Function Description** |
| GpiPolylineDisjoint | Draw a series of disjoint straight lines using the end-point pairs specified |

## Transformations

| Table 4-26. Transformation Functions | |
|---|---|
| **Name** | **Function Description** |
| GpiConvertWithMatrix | Convert an array of coordinate pairs from one coordinate space to another, using the supplied transform matrix |

## PM Helper Macros

A set of helper macros is defined in the PMWIN.H header files for OS/2 2.0 PM. These macros simplify programming for button, list-box, and menu controls. Previously, programmers had to set up complex parameters to send messages to the control windows in order to modify list-box items, menu items, and so on. The new macros provide the code expansions for these complex functions. Table 4-27 on page 4-53 lists the new helper macros.

*Table 4-27. PM Helper Macros*

| Name | Function Description |
|---|---|
| WinCheckButton | Set the checked state of the specified button control |
| WinCheckMenuItem | Set the check state of the specified menu item |
| WinDeleteLboxItem | Delete an indexed item form the List Box |
| WinEnableControl | Set the enable state of the item in the dialog template |
| WinEnableMenuItem | Set the state of the specified menu item |
| WinInsertLboxItem | Insert text into a List Box at index |
| WinIsControlEnabled | Return the state of the specified item in the dialog template |
| WinIsMenuItemChecked | Return the state of the identified menu item |
| WinIsMenuItemEnabled | Return the state of the menu item specified |
| WinIsMenuItemValid | Return TRUE if the specified item is a valid choice |
| WinQueryButtonCheckState | Return the checked state of the button control specified |
| WinQueryLboxCount | Return the number of items in the List Box |
| WinQueryLboxItemText | Fill the buffer with the text of the indexed item |
| WinQueryLboxItemTextLength | Return the length of the text of the indexed item in the List Box |
| WinQueryLboxSelectedItem | Return the index of the selected item in the List Box |
| WinSetLboxItemText | Set the text of the List Box indexed item to buffer |
| WinSetMenuItemText | Set the text for Menu indexed item to buffer |

## Summary

Many Control Program functions have been renamed, replaced, or enhanced. The new guidelines used to name functions ensure compliance with Get, Set, and Query semantics used in PM-SAA conventions, the use of action verbs before nouns, and the use of consistent semantics for similar actions. Some 16-bit functions have been redesigned for the 32-bit environment, in particular memory-management, semaphore, and signal functions.

Control Program functions that have changed in the 32-bit version of the operating system include:

- Memory management functions
- Thread and Process functions
- Semaphore functions
- Pipe, queue, and timer functions
- Dynamic linking functions
- Device I/O functions
- File system functions
- Message retrieval functions
- Code-page management functions
- Session management functions
- Error management functions
- Signal functions
- Exception management functions
- VDD services functions

Many 16-bit PM functions have been replaced by new functions in the 32-bit function set, while others are no longer available. The functions which are no longer available affect the following areas:

- Heap management
- Installed program list
- Initialization file
- Window locking

New functions are available for:

- Printing
- Workplace
- Customizing help information
- 32-bit migration
- Standard dialogs
- Pop-up menus
- Desktop background
- Paths, regions, and bit maps
- Fonts and characters
- Polylines
- Transformations

In addition, new window controls, hook capabilities, and helper macros are provided.

# Chapter 5. Dynamic Linking

This chapter describes:

- Static vs. Dynamic Linking
- Dynamic link library Data
- Dynamic link library Initialization and Termination
- Building a dynamic link library (DLL)
- Use of protected memory by DLLs

## Static vs. Dynamic Linking

Most programmers are familiar with static linking; an application calls a routine or procedure whose code is not found in the application's source file. The routine is *external* to your source file and is declared as such. When the source file is compiled, the compiler places an external reference for the routine in the application's object file. To create the executable file for the application, the application's object file is linked with an object file that contains the code for the routine. The result is an .EXE file that contains the application code, as well as a copy of the code for the routine. Figure 5-1 illustrates the process of building a statically-linked application.

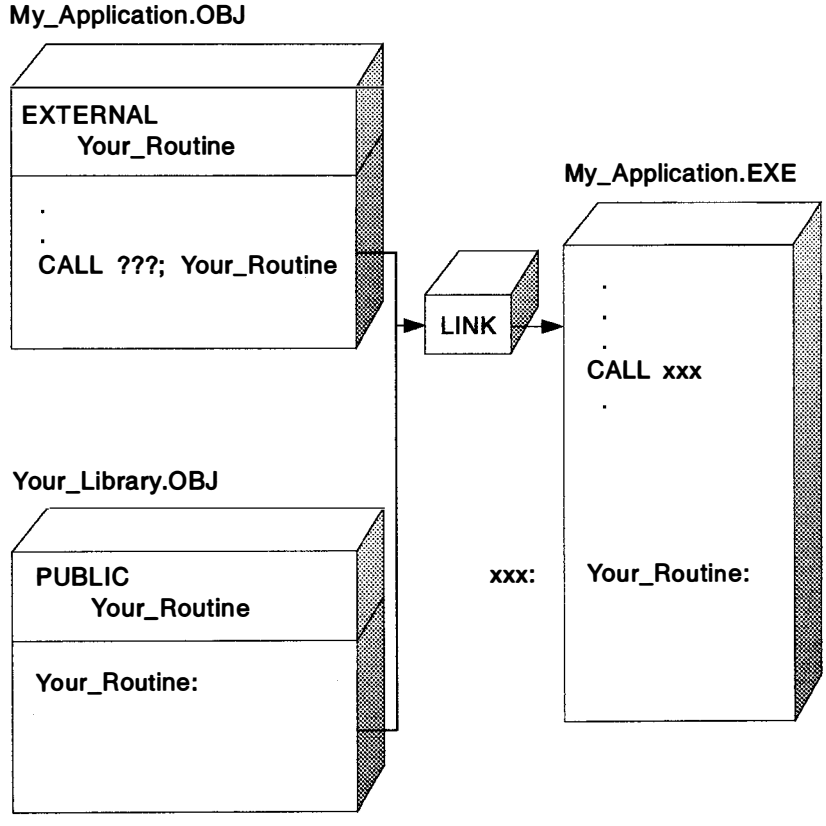



*Figure 5-1. Static Linking*

When OS/2 2.0 loads a statically-linked program, all the code and data are contained in a single executable file and the system can load it all into memory at once.

The advantages and disadvantages of static linking are summarized in Table 5-1

*Table 5-1. Why Static Linking?*

| Advantages | Disadvantages |
|---|---|
| Compile in pieces | External routines built into EXE (making EXEs larger) |
| Can create libraries of routines that can be linked with applications | EXE cannot be changed without re-linking |
| | External routines cannot be shared (duplicate copies of libraries) |

Dynamic linking allows several applications to use a single copy of an executable module. The executable module is completely separate from the applications that use it. Several functions can be built into a DLL, and applications can use these functions as if they were part of the application's executable code. You can change the dynamically-linked functions without recompiling or relinking the application.

The advantages of dynamic linking are:

**Reduced memory requirements**

Many applications can use a single DLL simultaneously. Since only one copy of the DLL is in memory, this saves memory space, and in general, the code is discardable.

**Simplified application modification**

An application can be enhanced or modified by simply changing a DLL. For example, if an application uses a DLL to support video output, several displays can be supported by different DLLs. The application can use the DLL that supports the appropriate display.

**Flexible software support**

DLLs can be used for after-market support. In the display-driver example, a new DLL can be provided to support a display that was not available when the application was shipped. Similarly, a database application can support a new data-file format with a new DLL.

**Transparent migration of function**

The DLL functions can be used by applications without any understanding of how the functions actually do their work. Future changes to the DLL are transparent to the application, as long as the input and output parameters remain the same.

**Multiple programming language support**

A function in a DLL can be used by an application written in any programming language, as long as the application understands the function's calling convention.

**Application-controlled memory usage**

Applications can make decisions about which DLL routines they wish to load into memory and use. If a DLL is not used, it does not have to be loaded.

DLLs can be used to implement subroutine packages, subsystems, and interfaces to other processes. In OS/2 2.0, some DLLs:

- Are interfaces to the kernel

  The worker routines for the OS/2 API reside in the OS/2 kernel. Applications, which run at privilege level 3, can usually make direct calls to the kernel, which runs at privilege level 0. Some calls, however, must be linked through a DLL. For example, an application that calls DosOpen is linked to the DLL DOSCALL1.DDL. This library contains the definitions for some system functions. When a system function is called, OS/2 2.0 makes the call to the kernel on behalf of the application.

- Are interfaces to devices

  DLL subsystems can virtualize devices and manage the device for its clients.

- Provide an open system architecture for software

  Add-ons to OS/2 2.0 can be provided easily and by anyone.

OS/2 2.0 provides two varieties of dynamic linking: *load-time* and *run-time*. In load-time dynamic linking, references are resolved when an application is loaded. In run-time dynamic linking, references are resolved when the application runs.

## Load-Time Dynamic Linking

Load-time dynamic linking is similar to static linking in that an application can call a routine that is not found in the application's source file. In load-time dynamic linking, however, an application is linked with a library file that contains a record that describes where the routine can be found instead of with a file that contains the code for the routine. The resulting application executable file contains this record and not a copy of the routine's code. Figure 5-2 on page 5-4 illustrates the process of building a load-time dynamically-linked application.

In the example in Figure 5-2 on page 5-4, the .LIB file contains a record that describes where the code for a set of functions can be found. In this case, the code for the function Your_Routine can be found in the file, or module, Your_Routines.DLL under the entry point name Your_Routine. (The entry point name does not have to match the function name.) The function code can also be referenced by its ordinal value.

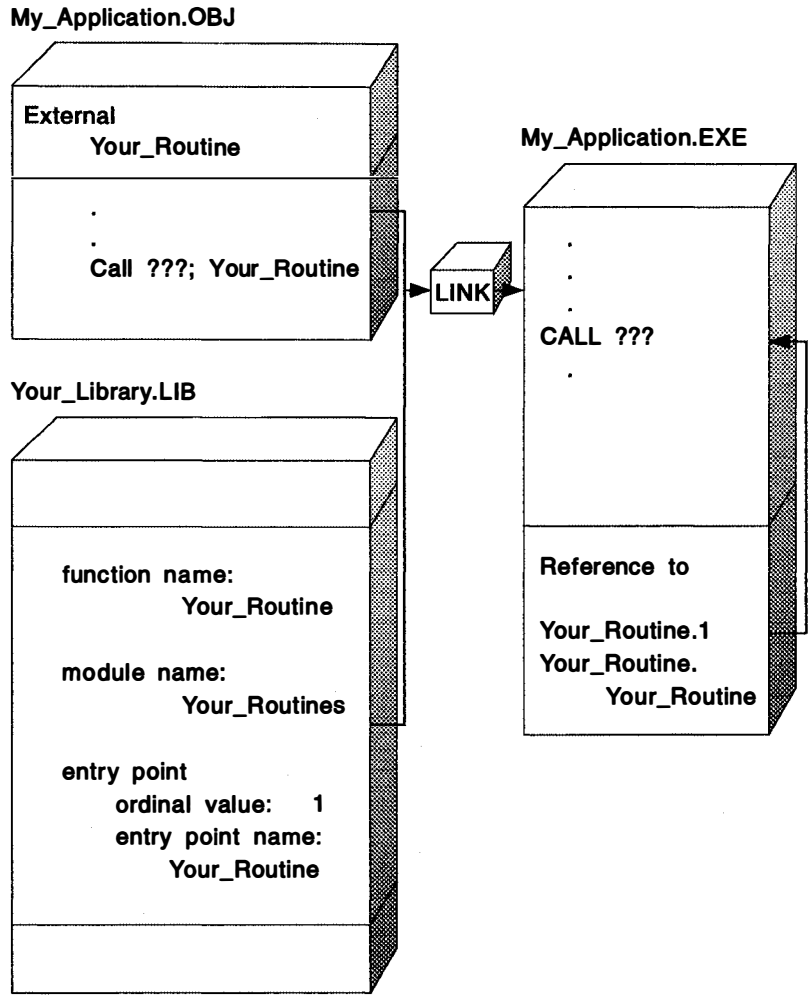



*Figure 5-2. Dynamic Linking*

When the application is loaded, the system resolves the dynamic-link references, as shown in Figure 5-3 on page 5-5.

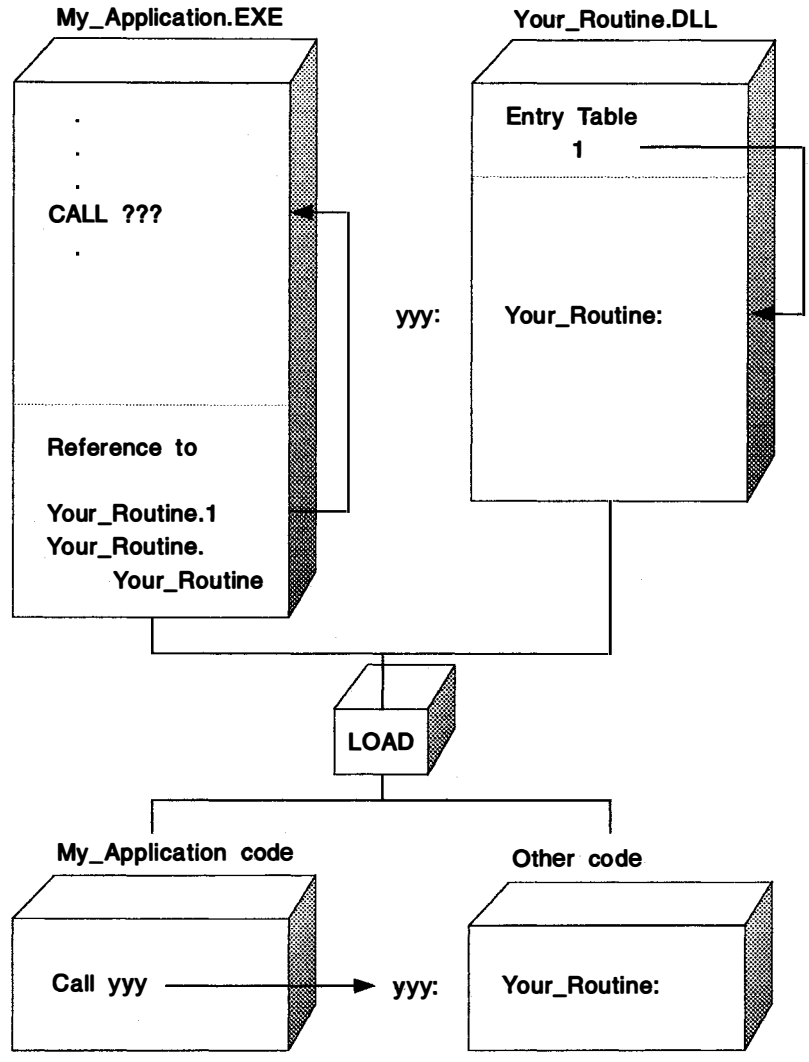



*Figure 5-3. Resolving Dynamic Link References*

If a program contains dynamically-linked references, the system must process the information in the references. If the DLL is already in memory, the system simply adds information to the executable code so that the code can use the DLL functions. If the required DLLs are not already in memory, OS/2 2.0 searches the path specified by the LIBPATH environment variable. If the system cannot find the DLLs, it stops loading the application and reports the error. If the system finds the DLLs, it loads them. When DLLs are loaded into memory, OS/2 2.0 notifies the application where the DLL functions can be found.

When a DLL is loaded into memory is determined by how the DLL was built. A DLL is built like an application, using a module-definition (.DEF) file. The CODE statement in a .DEF file describes the attributes of application or DLL code. The *load* option for the CODE statement determines when application or DLL code is loaded:

**PRELOAD** Code is loaded as soon as a process accesses the DLL. This leads to increased performance (in terms of accessing the DLL functions) while decreasing available memory. This option might be preferable if there is no more than one program running.

**LOADONCALL** Code is loaded when needed. This is the recommended choice, and the default.

## Run-Time Dynamic Linking

When an application contains a reference to a DLL, the DLL is loaded into memory when the application is loaded (unless the DLL is already in memory). If the application uses functions in several DLLs, all of those DLLs are loaded into memory. Some applications may use functions in several DLLs but use only a few of them at any one time. For example, an application that supports many different printers or plotters may use functions in many DLLs but need only one of them at a time, depending on the printer or plotter the application is supporting. If the user switches to a different printer or plotter, another DLL will be used, but the first will remain in memory. Loading DLLs this way can be very wasteful.

To avoid this problem, applications can take advantage of run-time dynamic linking in OS/2 2.0 and load and unload DLLs as they are required. The process of building a run-time dynamically linked application is similar to the process of building a load-time dynamically linked application. However, the .EXE for a run-time dynamically linked application does not contain a record describing where the external routines can be found. Instead, the application explicitly tells OS/2 2.0 when to load and free the dynamic link module.

Applications load and unload DLLs and call functions whose code reside in those DLLs by:

1. Calling DosLoadModule to get a handle to the DLL module.

   This function also loads the DLL into memory, or attaches to it, if it is already loaded.

2. Calling DosQueryProcAddr to get a pointer to a function within the DLL.
3. Calling the function indirectly through the pointer returned by DosQueryProcAddr.
4. Calling DosFreeModule to free the handle to the DLL module.

   When all handles to the DLL module have been freed, the DLL is unloaded from memory.

An application can also request information about a DLL from the system. The application can use the DosQueryModuleHandle function to determine whether a DLL has already been loaded into memory. The DosQueryModuleName function returns full path information for the DLL file.

The advantages of run-time dynamic linking are:

**Memory is consumed as needed.**

DLLs can be loaded and unloaded as they are used. Unused DLLs do not have to be loaded into memory, and memory can be released when the application has finished using the DLL.

**Applications can recover from DLL NOT FOUND**

Applications can make decisions. If a load-time DLL cannot be found, the application terminates immediately. If a run-time DLL cannot be found, the application receives an error value from the DosLoadModule function, and it can then use another DLL or specify a full path for the DLL. If a function is not available, the DosQueryProcAddr function returns an error value, and the application can take appropriate action.

**DLL and function names can be variable.**

An application programmer does not have to know the names of the DLLs the application will be using or the names of the functions within the DLL. The application can read the names of the DLL or the functions from a configuration file or obtain the names from user-supplied input.

**DLLs can be anywhere.**

The application can specify the full path for the DLL. Load-time DLLs must be in a directory in the path specified by the LIBPATH environment variable, but run-time DLLs can be in other directories.

## DLL Data

Most DLLs will contain some data. Whereas DLL code is shared by all processes that use it, DLL data can be shared or not shared by all processes that use it. Data that is specific to each process that uses the DLL (that is, to each instance of the DLL) is called *instance data*. Data that is shared by all processes that use the DLL is called *shared* or *global data*.

The first time a process references a DLL (and it is loaded or its usage count is incremented because it is already loaded), a separate copy of the DLL's instance data is created. Modifications to the instance data for one process do not affect the instance data for any other process. The system, however, maintains only one copy of a DLL's shared data for all processes that use the DLL. Every process that uses the DLL can access the same data. If one process modifies the data (increments a count, for example), the data will be modified for all processes that use the DLL.

Because changes to shared DLL data by one process are visible to the DLL code when called by another process, shared data provides DLLs with a mechanism for tracking processes that use it. This is crucial to subsystems, which are DLLs that manage resources (for example, devices, queues, and so on).

There is usually only one data and one code object, or segment, defined in a C source file. This means that a DLL that has instance and shared data is built from more than one C source file, with a default automatic data segment and with named data segments. How data is defined is determined by how the DLL is built. A DLL is

built like an application, using a .DEF file. The DATA statement in a .DEF file describes the attributes of application or DLL data. Following is a list of the available options for the DATA statement:

| Options | Result |
|---|---|
| **MULTIPLE** | OS/2 2.0 creates a unique copy of the automatic data segment for each process that uses the DLL. Modifications made by one process do not affect any other process. This is the default. |
| **SINGLE** | OS/2 2.0 creates only one automatic data segment for all processes that use the DLL. If one process modifies the data, the data will be modified for all processes that use the DLL. |
| **READWRITE** | Enables you to read from or write to the automatic data segment. This is the default. |
| **READONLY** | Enables you to only read from the automatic data segment. |
| **LOADONCALL** | The automatic data segment is loaded into memory as needed. This is the default. |
| **PRELOAD** | The automatic data segment is loaded as soon as a processes accesses a DLL. |

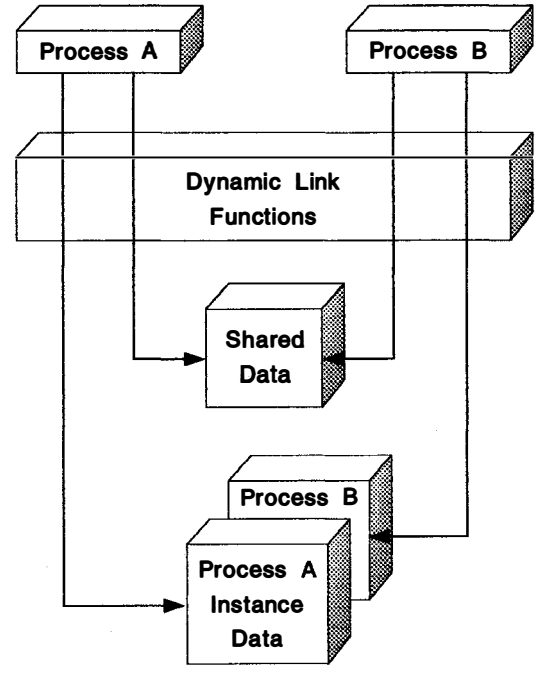



*Figure 5-4. DLL Data*

You can also create a DLL with both shared and instance data. For more information, see "Using Shared and Instance Data" on page 5-13

## DLL Initialization and Termination

A DLL is initialized and terminated by the default _DLL_InitTerm function. When a process gains access to the DLL, this function initializes the necessary environment for the DLL, including storage, semaphores, and variables. When the process frees its access to the DLL, the _DLL_InitTerm function terminates the DLL environment created for that process. The _DLL_InitTerm function is called automatically when you link to the DLL.

The _DLL_InitTerm function can be executed once, when the DLL is first loaded into memory, or it can be executed each time a new process first accesses the DLL. The LIBRARY statement in the module-definition file is used to specify when the _DLL_InitTerm function is to be executed. Following is a list of of the available options for the LIBRARY statement.

| Options | Result |
|---|---|
| **INITINSTANCE** | The _DLL_InitTerm function is called the first time the DLL is loaded for each process that accesses the DLL. |
| **INITGLOBAL** | The _DLL_InitTerm function is called only the very first time the DLL is loaded. This is the default. |
| **TERMINSTANCE** | The _DLL_InitTerm function is called the last time the DLL is freed for each process that accesses the DLL. |
| **TERMGLOBAL** | The _DLL_InitTerm function is called only the final time the DLL is freed. This is the default. |

As an example, the following statement,

```
LIBRARY SAMPLE03 INITINSTANCE TERMINSTANCE
```

*Figure 5-5. Specifying when to Execute the Initialization and Termination Function*

identifies the executable file as a DLL and specifies that SAMPLE03 is the name of the DLL. It also specifies that the _DLL_InitTerm function is to be called the first time the DLL is loaded for each process that calls the DLL and the last time the DLL is freed for each process that calls the DLL.

When OS/2 2.0 starts executing a DLL, it sets the CPU registers to known values, but only for 16-bit DLLs. All 32-bit DLLs are called with a stack frame, like all other API calls.

Initialization/termination functions can be written in a high level language. For more information on writing your own initialization/termination function, see "Creating an Initialization/Termination Function" on page 5-13.

## Building DLLs

Building a DLL is not very different from building a conventional static library. The following sections show how you can use OS/2 tools and functions to create, manage, and use DLLs.

## External Function References

When you compile or assemble an application, the compiler or assembler generates an object module for the code in the application. If you use any functions that are external to your application (their code is in another object module), the compiler or assembler adds an external function reference to your application's object module.

The Linker resolves these external references. If the Linker finds the external function in a library called an import library or in an IMPORTS statement in the application's module-definition file, the code for the external function is in a DLL. To resolve external references to DLLs, the Linker simply adds information to the executable file that tells the loader where to find the DLL code when the application is loaded.

## Module-Definition Files

The module-definition file is an important tool for building DLLs. This file contains information that tells OS/2 2.0 the name of the DLL, when to load the DLL, how to manage memory for the DLL, and when to initialize the DLL.

When you create a DLL, the module-definition file must contain a list of all the functions in the DLL that can be called by an application (or by another DLL). You specify these external functions by using an EXPORTS statement in the module-definition file.

You must also tell the Linker where to find the external functions in your application. If the functions are in a DLL, you can use an IMPORTS statement in the module-definition file for the application to tell the Linker where to find the DLL functions. You can also use an import library to tell the Linker where to find your DLL functions.

## Import Libraries

A conventional library contains object modules for a number of functions. The library is a convenient way to manage a large number of modules and use them in your executable code by linking to the library. The Linker uses the external references in your object module to determine which modules must be pulled out of the library.

An import library does not contain any object modules. Instead, the import library contains information that tells the Linker what DLLs are used by your application and the location of the functions your application uses within each DLL.

Like a conventional library, an import library is primarily a convenience. Instead of specifying all the functions your application imports in its module-definition file, you can simply link with the import library and let the Linker resolve the external references in your object module.

You use import libraries every time you compile and link a program that uses the OS/2 API. All the OS/2 functions are implemented in DLLs, and OS2386.LIB is simply an import library that tells the Linker where to find each OS/2 function.

For more information about module-definition files and import libraries, see the online Tools Reference in the OS/2 2.0 Developer's Toolkit.

## Creating a Simple DLL

DLLs are typically used to provide common functions that can be used by a number of applications. Figure 5-6 shows a C source file, MYPUTS.C, for a DLL that contains a simple string-printing function.

```
#include <os2.h>

#define HF_STDOUT 1  /* standard-output handle                    */

VOID EXPENTRY myPuts(PSZ pszMsg)
{
    ULONG cbWritten;

    while(*pszMsg) {
        DosWrite(HF_STDOUT, pszMsg, 1, &cbWritten);
        pszMsg++;
    }

}
```

*Figure 5-6. A Simple DLL*

Figure 5-7 shows the module-definition file, MYPUTS.DEF, for this DLL.

```
LIBRARY myputs
DATA SINGLE SHARED
EXPORTS
    myPuts
```

*Figure 5-7. Module-Definition File for Simple DLL*

The LIBRARY statement names the DLL (MYPUTS.DLL). The DATA statement tells the system that this DLL will share all data with each process that uses the DLL. The EXPORTS statement indicates that the function myPuts can be used by applications and DLLs.

The DLL is compiled and linked like any application. You can use IBM's ICC and LINK386, as shown below, to create MYPUTS.DLL.

```
icc  /C+ /Ge- myputs.c

link386 /noi myputs, , nul, OS2386, myputs
```

*Figure 5-8. Compiling and Linking a Simple DLL*

When the DLL has been created, you must copy it to one of the directories indicated by the LIBPATH environment variable in your CONFIG.SYS file.

## Importing DLL Functions

After you create a DLL, you can use it in an application. Figure 5-9 shows a C source file, USEPUTS.C, that uses the myPuts function contained in the DLL MYPUTS.DLL.

```
#include <os2.h>

VOID EXPENTRY myPuts(PSZ);

VOID main(VOID)
{
    myPuts("Testing, 1,2,3");
}
```

*Figure 5-9. Using a Simple DLL*

The module-definition file for USEPUTS.C tells OS/2 2.0 where to find the myPuts function. This module-definition file (USEPUTS.DEF) contains the information shown in Figure 5-10.

```
NAME useputs
IMPORTS
    myputs.myPuts
```

*Figure 5-10. DEF File for Application Using a DLL*

The module-definition file tells OS/2 2.0 that USEPUTS imports the myPuts function from MYPUTS.DLL. USEPUTS.C is compiled and linked as shown below.

```
icc  /C+ useputs.c

link386 /noi useputs, , nul, , useputs
```

*Figure 5-11. Compiling and Linking an Application*

## Using an Import Library

You can also create an import library for your DLL. If you do this, you can link applications with your DLL without explicitly declaring the imports for each application. OS/2 2.0 uses this technique for the application programming interface (API). When you link your applications with OS2386.LIB, you are using an import library.

To create the import library STRINGS.LIB from MYPUTS.DLL, you use the Import Library Manager (IMPLIB), as shown below.

```
implib strings.lib myputs.def
```

*Figure 5-12. Creating an Import Library*

You can then link your applications with STRINGS.LIB to resolve references to the myPuts function, as shown below.

```
link386 /noi useputs, , nul, strings;
```

*Figure 5-13. Linking with an Import Library*

Note that a module-definition file for USEPUTS.C is optional in this example, because we are linking with an import library.

## Using Shared and Instance Data

When you create a DLL, you can use the DATA statement in the module-definition file to define the default attributes for data segments within the DLL. The default condition is for the DLL to have a unique copy of the automatic data segment for each process. You can specify DATA MULTIPLE READWRITE in the module-definition file to cause OS/2 2.0 to create a separate copy of all the DLL data for each process that uses the DLL (instance data). Modifications made by one process do not affect other processes.

You can also specify different attributes for different sets of data by using the #pragma data_seg and #pragma alloc_text directives to define your own data and code segments. You can list the segments in the module-definition file under the heading SEGMENTS, and specify attributes for each.

```
SEGMENTS
   mydata SINGLE READONLY
   mycode PRELOAD
```

*Figure 5-14. Specifying Data Segments with Different Attributes*

Any segments that you do not specify under SEGMENTS are given the attributes specified by the DATA or CODE statement, depending on the type of segment.

## Creating an Initialization/Termination Function

It may be necessary for a DLL to perform some tasks before an application accesses a DLL or after an application finishes accessing a DLL. For example, the library may need to allocate a heap or open a device prior to using a DLL or deallocate a heap or close a device after using a DLL. You can handle these tasks in an initialization/termination function. The initialization/termination function can be called on to perform initialization tasks when the DLL is first loaded or each time a new process accesses the DLL, depending on the LIBRARY statement in the module-definition file. If you specify INITGLOBAL in the LIBRARY statement, the initialization/termination function is called only once, when the DLL is first loaded into memory. This is the default setting. If you specify INITINSTANCE, the library function is called each time the DLL is accessed by a new process. In the same way, the initialization/termination function can be called on to perform termination tasks. If you specify TERMGLOBAL in the LIBRARY statement, the initialization/termination function is called only once, the final time the DLL is freed. This is the default setting. If you specify TERMINSTANCE, the library function is called each time the DLL is freed for each process that accesses the DLL.

When a thread calls DosLoadModule to load a DLL, the initialization routines of the loaded DLL (and the initialization routines of the DLLs that it loads) will run on the thread that called DosLoadModule. This initialization will complete before DosLoadModule returns.

The prototype for the _DLL_InitTerm function is:

```
unsigned long _DLL_InitTerm (unsigned long modhandle, unsigned long flag);
```

*Figure 5-15. Prototype for the _DLL_InitTerm Function*

If the value of the *flag* parameter is 0, the DLL environment is initialized. If the value of the *flag* parameter is 1, the DLL environment is ended. The *modhandle* parameter is the module handle assigned by the operating system for this DLL. The module handle can be used as a parameter to various OS/2 API calls. For example, DosQueryModuleName can be used to return the fully-qualified path name of the DLL, which tells you where the DLL was loaded from.

The return code from _DLL_InitTerm tells the loader if the initialization or termination was performed successfully. If the call was successful, _DLL_InitTerm returns a nonzero value. A return code of 0 indicates that the function failed. If a failure is indicated, the loader will not load the program that is accessing the DLL.

Before you can call any C library functions, you must first initialize the C run-time environment. To initialize the environment, use the function _CRT_init. The prototype for this function is:

```
int _CRT_init (void);
```

*Figure 5-16. Prototype for the _CRT_init Function*

If the run-time environment is successfully initialized, _CRT_init returns 0. A return code of -1 indicates an error. If an error occurs, an error message is written to file handle 2, which is the usual destination of stderr.

To properly terminate the C run-time environment, use the function, _CRT_term. The prototype for this function is:

```
void _CRT_term (void)
```

*Figure 5-17. Prototype for the _CRT_term Function*

Because _DLL_InitTerm is called by the operating system, it must be compiled using the system linkage. In the IBM C Set/2 compiler, the following #pragma directive is used to specify the system linkage:

```
#pragma linkage (_DLL_InitTerm, system)
```

*Figure 5-18. Specifying the System Linkage*

The initialization/termination function must have a specific entry point. You cannot create a function with a specific entry point in the C programming language, so the initialization function must be written in assembly language. You can, however, write a very simple initialization function in assembly language and have it immediately jump to a C function. Figure 5-19 shows an assembly language initialization function entry point.

```
PAGE    ,132
TITLE   DLLSTUB
NAME    DLLSTUB

.386
.387

EXTERN  _DLL_InitTerm, PROC

END     _DLL_InitTerm
```

*Figure 5-19. A DLL Initialization Function Entry Point*

The following figure shows a sample initialization/termination function written in C. This code was written using the IBM C Set/2 compiler. If you use another compiler, some of the #pragmas or keywords may need to be changed.

```
/*+------------------------------------------------------------------+*/
/*| Sample Program 03 : INITTERM.C                                   |*/
/*|                                                                  |*/
/*| COPYRIGHT:                                                       |*/
/*| ----------                                                       |*/
/*| Copyright (C) International Business Machines Corp., 1991        |*/
/*|                                                                  |*/
/*+------------------------------------------------------------------+*/

#include <stdlib.h>
#include <stdio.h>

/*+------------------------------------------------------------------+*/
/*| _CRT_init is the C run-time environment initialization function. |*/
/*|It will return 0 to indicate success and -1 to indicate failure.  |*/
/*+------------------------------------------------------------------+*/

int _CRT_init (void);
```

*Figure 5-20 (Part 1 of 3). A DLL Initialization Function*

```
/*+-------------------------------------------------------------------+*/
/*| _CRT_term is the C run-time environment termination function.     |*/
/*+-------------------------------------------------------------------+*/

void _CRT_term (unsigned long);

size_t nSize;
int *pArray;

/*+-------------------------------------------------------------------+*/
/*| _DLL_InitTerm is the function that gets called by the operating   |*/
/*| system loader when it loads and frees this DLL for each process   |*/
/*| that accesses this DLL.  However, it only gets called the first   |*/
/*| time the DLL is loaded and the last time it is freed for a        |*/
/*| particular process.  The system linkage convention must be used   |*/
/*| because the operating system loader is calling this function.     |*/
/*+-------------------------------------------------------------------+*/

#pragma linkage (_DLL_InitTerm, system)

unsigned long _DLL_InitTerm (unsigned long modhandle, unsigned long flag)
   {
   size_t i;

   /* If flag is zero then the DLL is being loaded so initialization  */
   /* should be performed.  If flag is 1 then the DLL is being freed  */
   /* so termination should be performed.                             */

   switch (flag)
      {
      case 0:
         /* The C run-time environment initialization function must   */
         /* be called before any calls to C run-time functions that   */
         /* are not inlined.                                          */

         if (_CRT_init () == -1)
            return 0UL;

         srand (17);

         nSize = (rand() % 128) + 32;

         printf ("The array size for this process is %u\n",nSize);
```

*Figure 5-20 (Part 2 of 3). A DLL Initialization Function*

```
        if ((pArray=malloc (nSize * sizeof(int)))==NULL)
           {
           printf("Could not allocate space for unsorted array.\n");
           return 0UL;
           }

        for (i=0; i<nSize; ++i)
           pArray[i] = rand();

        break;

     case 1:
        printf("The array will now be freed.\n");

        free(pArray);

        _CRT_term(0UL);

        break;

     default:
        printf("flag = %lu\n",flag);
        return 0UL;

     }

  /* A non-zero value must be returned to indicate success. */
  return 1UL;

  }
```

*Figure 5-20 (Part 3 of 3). A DLL Initialization Function*

You can also write the initialization/termination function entirely in assembly language, without jumping to a C function. For this case, the library initialization registers are defined as follows:

| | |
|---|---|
| **EIP** | Library entry address |
| **ESP** | User program stack |
| **CS** | Code selector for base of linear address space |
| **DS = ES = SS** | Data selector for base of linear address space |
| | **Note:** All 32-bit protected memory library modules will be given a GDT selector in the DS and ES registers (ProtDS) that addresses the full linear address space available to an application. This selector should be saved by the initialization routine.<br>Non-protected memory library modules will receive a selector (FlatDS) that addresses the same amount of linear address space as an application's .EXE file. |
| **FS** | Data selector of the base of the Thread Information Block (TIB) |
| **GS** | Is equal to 0 |
| **EAX = EBX** | Is equal to 0 |
| **ECX = EDX** | Is equal to 0 |

| | |
|---|---|
| **ESI=EDI** | Is equal to 0 |
| **EBP** | Is equal to 0 |
| **[ESP+0]** | Return address to system, and EAX = return code |
| **[ESP+4]** | Module handle for the library module |
| **[ESP+8]** | Is equal to 0 (for initialization) |

A 32-bit library may specify that its entry point address is the 16-bit object code. In this case, the entry registers are the same as for entry to a library using the segmented .EXE format. This means that a 16-bit library may be relinked to take advantage of the benefits of the linear .EXE format (such as more efficient paging).

The library termination registers are defined as follows:

| | |
|---|---|
| **EIP** | Library entry address |
| **ESP** | User program stack |
| **CS** | Code selector for the base of the linear address space |
| **DS=ES=SS** | Data selector for the base of the linear address space |
| **FS** | Data selector of the base of the Thread Information Block (TIB) |
| **GS** | Is equal to 0 |
| **EAX=EBX** | Is equal to 0 |
| **ECX=EDX** | Is equal to 0 |
| **ESI=EDI** | Is equal to 0 |
| **EBP** | Is equal to 0 |
| **[ESP+0]** | Return address to the system |
| **[ESP+4]** | Module handle for the library module |
| **[ESP+8]** | Is equal to 1 (for termination) |

**Note:** Library termination is not allowed for libraries with 16-bit entries

## Linking at Run Time

So far, the examples in this chapter have used load-time dynamic linking. With load-time linking, OS/2 2.0 loads the DLL containing the imported functions when it loads the .EXE file. If OS/2 2.0 cannot find the necessary DLL, it terminates the application and reports the error.

Run-time dynamic linking allows an application to load a DLL into memory when it is required, and to remove the DLL when it is no longer needed. The application uses the DosLoadModule function to load the DLL into memory (if it is not already loaded). If the system cannot find the DLL, the application receives an error value and can take appropriate action (for example, the application might use another DLL or search another directory).

Once the application has loaded the DLL, it can use the DosQueryProcAddr function to obtain a pointer to the required function (or functions). The application can then use the function. When the DLL is no longer required, the application can use the DosFreeModule function to remove the DLL from memory. If there are other applications using the DLL, it remains in memory until the last application frees the DLL.

An application can specify a full path for the run-time DLL. If you specify the full path name, you can have two DLLs with the same name loaded at the same time, as in C:\OS2\DLLFILE.DLL and C:\OS2\DLL\DLLFILE.DLL. If the path is not specified, OS/2 2.0 assumes the DLL has the extension .DLL and looks for the file in the directories specified by the LIBPATH environment variable. Figure 5-21 uses the run-time dynamic-linking functions to access the myPuts function in the MYPUTS.DLL dynamic link library.

```
#define INCL_DOSMODULEMGR
#include <os2.h>

VOID (* EXPENTRY myPuts) (PSZ);

VOID main(VOID)
{
    HMODULE hmod;
    ULONG ulErr;
    UCHAR szFailName[CCHMAXPATH];

    ulErr = DosLoadModule(szFailName,          /* failed module name */
        sizeof(szFailName),                    /* size of buffer     */
        "myputs",                              /* name of DLL        */
        &hmod);                                /* module handle here */

    if (ulErr)
        DosExit(EXIT_PROCESS, 1);

    ulErr = DosQueryProcAddr(hmod,    /* DLL module handle           */
        0,                            /* function ordinal value      */
        "myPuts",                     /* function name               */
        (PFN *) &myPuts);             /* address of function pointer */

    if (!ulErr) {

        /* We can use the function now. */

        myPuts("does it work?");

        DosFreeModule(hmod);                  /* frees the DLL module */
    }
}
```

*Figure 5-21. Using A Runtime Dynamic-Linked Library*

## Protected Memory Use

OS/2 2.0 provides shared library support in the form of 32-bit DLLs. All 32-bit dynamic links or APIs are called using near CALL or RET instructions, so the cost of making dynamic-link calls should be significantly less than the cost of making the comparable calls in the 16-bit version of the operating system, where a far CALL is required. The DLLs execute in the context of the caller.

All 32-bit DLLs are mapped into the appropriate shared memory region of the requesting processes at load time and execute at ring 3 without IOPL. This model's protection characteristics correspond closest to the ring 3 dynamic linking model in

the 16-bit version of the operating system. Figure 5-22 on page 5-20 shows how 32-bit DLLs are implemented in the linear memory model of OS/2 2.0.

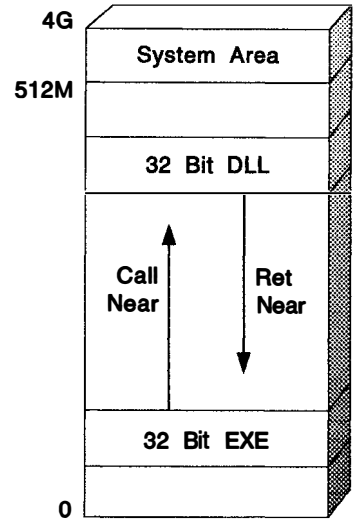



*Figure 5-22. A 32-Bit DLL*

However, since 32-bit EXE programs can address the entire address space with a 32-bit offset, it is easier for a 32-bit application programmer to potentially cast a bad pointer to data in the shared region than in the 16-bit segmented addressing scheme. Since many subsystems have semaphores and other shared data structures in the shared region, the potential for an inadvertently errant application to affect another process sharing a subsystem becomes an issue in the flat environment. Therefore, OS/2 2.0 provides a mechanism for DLLs to protect their critical shared global data regions from 32-bit EXEs. This mechanism prevents a thread in one process from potentially affecting other processes using the same resources (subsystems), or potentially taking down the entire workstation if the subsystem compromised is a critical subsystem (such as PM).

OS/2 2.0 provides the capability for existing 16-bit DLLs and new 32-bit DLLs to get their shared global data allocated into a single protected region that is not accessible by 32-bit EXEs, thereby achieving a level of protection. Note that there is no provision for protecting DLLs from each other or from threads executing 16-bit EXE modules. The MEMMAN CONFIG.SYS line supports a "PROTECT/NOPROTECT" option as follows for enabling or disabling memory protection:

```
MEMMAN=SWAP,PROTECT
```

*Figure 5-23. Enabling Memory Protection for DLLs*

If neither PROTECT or NOPROTECT is specified, the default is protection enabled (PROTECT).

When protection is enabled, the memory manager reserves a 64M region of the linear address space below the 512MB line. This region is called the *protected region*. Protected objects are allocated within the protected region. The following types of memory are considered protected:

**DLL Global Data** Global data that is part of the DLL image when loaded. This is only global shared data, not instance data. Although DLL code is shared, it is not allocated in the protected region since it is read-only.

**DLL Runtime Shared Data** Global data that is allocated at runtime by a thread executing in DLL code that is a protected API. This includes 16- and 32-bit, named and unnamed, shared memory, and shared memory allocated with DosAllocSeg with the share flag set.

The DS value that is used for the user address space (FlatDS) no longer references a descriptor with a 512MB limit. Instead the system exports another DS value for the user address space called the ProtDS that does have the 512MB limit—the FlatDS limit is reduced by the size of the protected region. When a 32-bit EXE is executing, it runs with the FlatDS and is unable to access protected objects created by 16-bit, 32-bit, or 16- and 32-bit DLLs. If the thread calls a 16-bit DLL API entry point, the DLL will have addressability to the protected region through the LDT. If the thread calls a 32-bit DLL entry point that is a protected one, the 32-bit DLL entry point contains code to switch to the ProtDS so that the protected region is accessible—the 32-bit DLL switches back to the FlatDS before completing service. A switch on the C compiler is used to generate the code sequence as shown in Figure 5-24.

```
DLLAPI  proc
    push    ds
    push    es
    mov     dx, seg FLAT:DGROUP
    mov     ds, dx
    mov     es, dx

    .....

    pop     es
    pop     ds
    ret
DLLAPI  endp
```

*Figure 5-24. Accessing Protected DLL Data*

Note that although SS is not loaded with the ProtDS, a subsystem that switches stacks to a protected stack must write some assembler code to change ESP—thus the subsystem should also setup SS to be the ProtDS when performing the stack switch.

When protection is not enabled, FlatDS = ProtDS and the code still works the same.

**Note:** The system is not currently sensitive to whether parameters are being validated relative to the FlatDS or the ProtDS when ring 0 kernel APIs are called. Also the 32→16 thunks do not probe 32-bit parameters before converting them and passing them to a 16-bit DLL.

The grouping of protected allocations can be enabled or disabled on a per DLL basis. For 32-bit DLLs, the Linker uses the PROTECT parameter in the .DEF file to provide protection information in the DLL's module flags to the loader. All 16-bit modules requiring protection must be specified with the new PROTECT16 CONFIG.SYS parameter.

```
PROTECT16=DLLNAME1,DLLNAME2,...,DLLNAMEX
```

*Figure 5-25. Using the PROTECT16 Parameter*

Note that the .DLL suffix is not required. Only .DLL files can get the protection.

## DLL Side Effects

Dynamic link routines are not processes. They run on the thread of the calling process and therefore don't own resources. Any resource that they obtain or use is owned by the calling process. Authors of DLLs should be careful not to needlessly allocate resources until the resource is required by the calling process to perform the requested function. They should also free the resource as soon as it is determined that the resource is no longer required.

A dynamic link routine that obtains and uses resources should attempt to minimize the use of a process resources. For example, stack space should be conserved. If an application redirects file handle 5 and calls a DLL entry that expects file handle 5 to be an open handle to an associated device driver, unexpected results will occur.

If the routine opens an abundance of file handles, it may consider increasing the maximum number of file handles, so that the process maximum is not exceeded. However, increasing the maximum number of file handles for a process also increases the maximum number of file handles for all processes created by the current process. This will cause additional memory to be consumed and could cause problems for an application that assumes a limit of 20 file handles. Also, it should be noted that applications have the ability to redirect file handles.

Dynamic link routines should also not make system calls that affect the calling process environment. If a DLL changes a process's current directory, another thread running under the same process could fail a file I/O call if it assumes a given working directory.

Applications and DLLs should not make calls to other DLLs, including system DLLs, within a critical section. Since DLLs can use semaphores to synchronize threads within a process or between processes, calling a DLL within a critical section could cause application deadlocks. This would occur if the DLL requests a semaphore on behalf of the calling thread and another thread within the process owns the semaphore. Because the calling thread is in a critical section and is the only thread within the process that is allowed to execute, the semaphore will never be freed, causing a deadlock.

## Summary

There are two types of linking: *static* and *dynamic*. Static linking enables a program's code and data to be contained in a single executable file, enabling the system to load it all into memory at once. Dynamic linking allows several applications to use a single copy of an executable module, since the executable module is completely separate from the applications that use it.

The advantage of dynamic linking are:

- Reduced memory requirements
- Simplified application modification
- Flexible software support
- Transparent migration of functions
- Multiple programming language support
- Application controlled memory usage

OS/2 2.0 provide two types of dynamic linking: *load-time* and *run-time*. In load-time dynamic linking, an application is linked with a library file that contains a record that describes where the routine can be found instead of with a file that contains the code for the routine. The DLL can be loaded as soon as a process accesses the DLL or when needed. In run-time dynamic linking, the EXE for an application does not contain a record describing where the external routines can be found. Instead, the application explicitly tells OS/2 2.0 when to load and free the dynamic link module.

DLL data can be shared or not shared by all processes that use it.

# Chapter 6. Multiple Virtual DOS Sessions

This chapter describes:

- The differences between the OS/2 1.X real mode DOS Box and 2.0 Enhanced DOS Session
- Virtual device driver architecture, and how the behavior of virtual device drivers are affected by Enhanced DOS Session operations.

## Overview

Since the introduction of OS/2 Version 1.0 in 1987, the DOS compatibility mode has remained relatively unchanged. The 16-bit OS/2 DOS Mode, based on the 80286 architecture, runs as a real mode task inside the protect mode environment of Version 1.X of the operating system. Enhanced DOS Sessions in OS/2 2.0 is a totally redesigned DOS compatibility environment, which extends the DOS compatibility features of the operating system by exploiting the virtual 8086 mode of the 80386 and 80486, resulting in a better implementation of DOS.

Most users considering a switch to OS/2 2.0 demand that the operating system provide a DOS compatible environment to run their favorite DOS applications. OS/2 2.0 enables users to not only use their favorite DOS application, but also to have multiple DOS sessions running concurrently.

With a design based on the Virtual 8086 mode, OS/2 2.0 maintains a protected system environment, even while running DOS applications. This enhances system integrity and makes it possible to multitask DOS applications with OS/2 applications, so DOS programs can continue to execute in the background. OS/2 2.0 also allows DOS applications to display in windows under the control of PM.

## Enhanced DOS Sessions

Version 1.X of the OS/2 operating system is based on the capabilities of the 80286 processor; therefore, the only practical way to run a DOS application is as a real mode task. This is primarily because in the 16-bit version of the operating system, DOS applications address code and data in memory using a segment:offset address format, based on the earlier 8088/8086 processors. OS/2 applications, however, are written to run in protect mode, and use the selector:offset address format. To run a DOS application simultaneously with many OS/2 applications, the system switches between real mode and protect mode when necessary.

OS/2 2.0 is designed to fully exploit the advanced features of the 80386 processor. A major innovation in the 80386 is support for executing multiple 8086 (or 8088) tasks within the 80386 protect mode environment. An 8086 task in this environment is called a Virtual 8086 (V86) task. In OS/2 2.0, this V86 task is a DOS Session, which runs as a single-threaded V86 mode process. Each DOS Session in OS/2 2.0 is managed as a single-process session. The OS/2 scheduler controls task-switching in much the same way as an OS/2 application process.

Figure 6-1 on page 6-3 provides a comprehensive view of the OS/2 2.0 structure with the Enhanced DOS Session kernel and Virtual Device Drivers (VDD) shown in relation to the OS/2 kernel and Physical Device Drivers (PDD). Key components of the OS/2 and Enhanced DOS Session kernel are shown, with a schematic of how these components interact with each other. Following is a description of some of these key components:

**DOS Session:** The instance of a single Virtual 8086 mode process running on an 80386 or 80486 processor, that is emulating all DOS operating system functions and providing a compatible environment. Several Enhanced DOS Sessions can be multitasked using the task switching features of the 80386 and compatibles.

**Enhanced DOS Session Kernel:** Composed of the following three major components, Enhanced DOS Session kernel controls the state and operation of multiple DOS Mode sessions (the actual number of sessions that you can create depends on the amount of space that you have available on your swapfile partition; if you have enough space available, you may be able to create more than 50 sessions):

- **DOS Session Manager**: The DOS Session Manager creates, initializes, and terminates DOS Sessions. The DOS Session Manager provides various system services to virtual device drivers. These services are known as Virtual Device Helper functions. The DOS Session Manager manages system resources (that is, memory, timers, semaphores, files) for all DOS Sessions.
- **8086 Emulation**: Performs 8086 instruction decoding, controls the 80386 processor's I/O Privilege Map for each DOS Session, manages the reflection of software interrupts for each DOS Session, routes IN/OUT instruction traps to the appropriate virtual device driver, and manages various control structures required by each virtual device driver.
- **DOS Emulation**: Emulates the function and operation of DOS on a per-DOS Session basis. Each DOS Session emulates an entirely independent instance of DOS. DOS services are emulated within the Enhanced DOS Session kernel, or by invoking protect mode services provided by the OS/2 kernel. For example, most DOS file I/O functions are provided by the OS/2 file system.

**Virtual Device Driver (VDD):** A VDD emulates an aspect of system function, typically I/O. A VDD manages the way in which multiple DOS Sessions access hardware I/O services. VDDs obtain and release system resources via the Virtal Device Helper (DevHlp) services provided by the Enhanced DOS Session kernel. A VDD can directly access an I/O control device, or may perform I/O through a physical device driver using a direct call interface. A VDD can also simulate hardware interrupts into one or many DOS Session processes. Finally, a VDD can provide the logic to emulate BIOS and other software interrupt functions.

With Figure 6-1 on page 6-3 as a point of reference, the following discussion highlights the advantages of Enhanced DOS Session and how OS/2 2.0 provides an improved DOS compatibility environment.

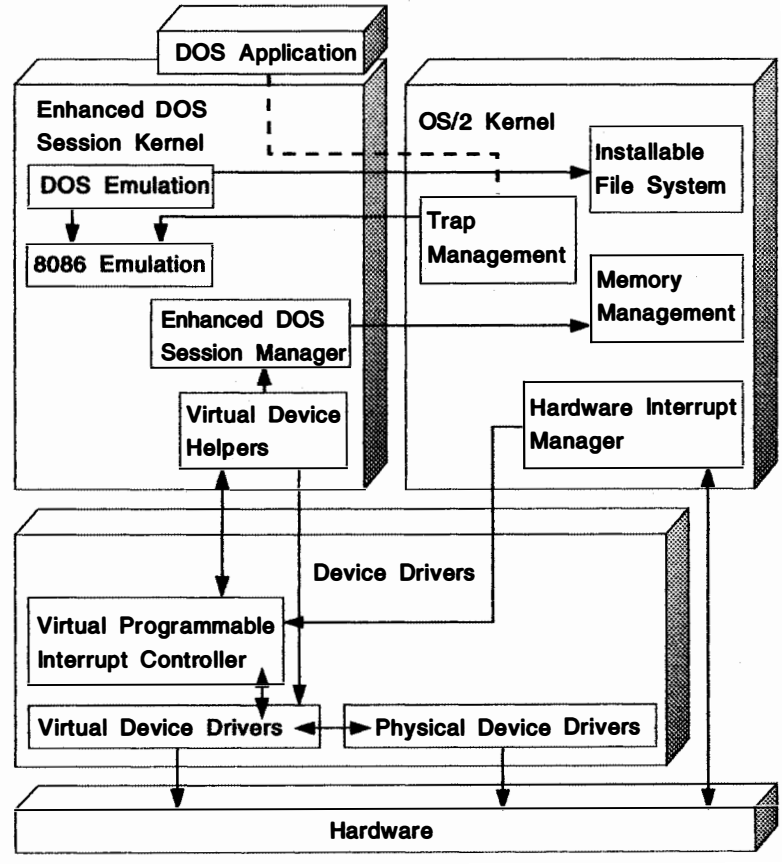



*Figure 6-1. Enhanced DOS Session System Structure and Control Flow*

## Fast Mode Switching

As mentioned previously, Version 1.X of the operating system is based on the 80286 processor. This processor can switch from real mode to protect mode, although this takes many CPU cycles. However, it has no provision for switching directly back to real mode. Switching back to real mode requires reinitializing the processor—in effect setting it to an initial power-on state. This must be done while preserving all the system control information, all process contexts, and system resource states. Thus, switching frequently between real mode, (where a DOS program is executing) and protect mode (where several OS/2 programs may be running), generates a significant level of system overhead.

Mode switching overhead is reduced on the 80386, where a single instruction switches the system quickly from real mode to protect mode and back again. Although this saves time, mode switching still requires significant overhead and is avoided whenever possible. For example, nearly all OS/2 1.X device driver code is bi-modal. That is, the device driver is written so that it can be entered in either

protect mode or real mode. An OS/2 bi-modal device driver will force a mode switch only when it is absolutely necessary, such as when it is running in real mode, but it must move data to a protect mode OS/2 application's virtual address space. When a DOS application is running while OS/2 applications run in the background, mode switching occurs. The overhead of mode switching can degrade system performance in these scenarios.

OS/2 2.0 DOS Sessions, on the other hand, are based on the Virtual 8086 mode of the 80386 and 80486 processors. The Virtual 8086 mode is a superset of protect mode, which is enabled by setting the VM bit in the EFLAGS register of the 80386. The CPU provides hardware support to switch automatically into V86 mode on a task switch when the new task sets the VM bit in the EFLAGS register. Because of this and other hardware support mechanisms, most of the operating system overhead associated with mode switching has been eliminated. When switching between V86 mode (DOS application) and protect mode (OS/2 application or system code), the 80386 hardware automatically activates the appropriate protection mechanisms.

Real mode execution has been totally eliminated in OS/2 2.0. This makes it possible to remove all real mode-specific code from device drivers and kernel modules, which were bi-modal in previous versions of the operating system. With OS/2 2.0, device drivers are written as purely protect mode executables. This simplifies the logic and reduces the size of the affected modules.

The OS/2 operating system provides each protect mode process with its own independent address space for code and data objects. This protects the code and data objects from other applications. It also helps prevent errant applications from causing a system crash, since system-owned resources are accessible only by system code. This high level of protection, a major feature of the OS/2 operating system not found in DOS, improves the integrity and reliability of the system.

Under Version 1.X of the operating system, when the OS/2 operating system switches into real mode, a DOS application can directly access any object in memory between 0 and 1MB, including portions of the OS/2 kernel and device drivers. A DOS program that inadvertently accesses any of the system's code or data objects can disrupt the system. In addition, a DOS program can directly access any I/O device hardware and cause the device to enter an unknown state. This can render the device useless in the OS/2 protect mode environment, and cause the OS/2 device driver to fail. These problems might eventually lead to a system crash and the loss of user data, which often accompanies such events.

By running DOS applications as V86 mode tasks, OS/2 2.0 maintains a fully protected system environment. Each DOS program runs in its own linear 1MB memory space. This space is separately allocated from system memory with full protection guaranteed by the 80386. A DOS program cannot corrupt any system code, data object, or another application's code or data. If the V86 task causes a trap or exception, it will be fully managed by the operating system to maintain integrity. An unpredictable DOS program can be terminated cleanly by the system in much the same manner as an unpredictable OS/2 application.

The system selectively isolates the DOS applications from I/O devices that are managed exclusively by OS/2 device drivers. These devices are then emulated, or virtualized, for one or more DOS applications. New to OS/2 2.0, the virtual device driver (VDD) provides each DOS application with a virtual instance of the real hardware, which is controlled by a physical device driver (PDD). If necessary for performance or other reasons, a VDD-PDD pair can cooperate to give the DOS program direct access to a particular I/O port or range of ports. By controlling

hardware access in this way, DOS applications cannot corrupt devices that are performing I/O functions for an OS/2 process.

## Multiple DOS Sessions

In OS/2 2.0, Enhanced DOS Sessions makes it possible to start many concurrent DOS sessions, each operating in its own independent 1MB linear address space. The number of sessions is only limited by the amount of memory and swap space. This brings true multiprogramming to the OS/2 DOS compatibility environment. The user can run multiple DOS programs in much the same way as running multiple OS/2 applications. DOS and OS/2 applications are started in the same ways, for example, from a command prompt, or from the Desktop.

## DOS Settings

In order to provide the highest possible level of compatibility with DOS applications, Enhanced DOS Sessions provide the user with the ability to customize the operation of the DOS environment through DOS Settings. DOS Settings allow the user to control particular DOS properties or attributes that affect the behavior of DOS applications running in a DOS Session. DOS applications typically are not careful about consuming system resources, such as memory and processor time. In order to preserve the integrity and performance of the system as a whole, Enhanced DOS Sessions provide a flexible environment for these applications by allowing the user to configure a DOS Session for DOS applications that might otherwise not work well (or not work at all) with the default settings of the DOS Session task.

Enhanced DOS Sessions provide a mechanism which supports standard settings, and allows virtual device drivers to register custom settings. The CONFIG.SYS file contains a number of standard DOS Settings. These are applied to all DOS Sessions as they are created; however, these settings can be changed for individual sessions.

DOS Settings can be set by the user when adding an application to a group on the desktop, or in certain cases, during execution while an application is running within the DOS Session. In the case where a DOS Session is created by another process using a DosStartSession function call, a buffer can be provided containing the required DOS Settings and their values.

The standard DOS settings that affect the operation of virtual device drivers supplied with OS/2 2.0 can be categorized as follows:

| | |
|---|---|
| **Video** | Control functioning of screen I/O operations within a DOS Session. |
| **Hardware Environment** | Affect the virtual hardware environment provided by the DOS Session. |
| **DOS Environment** | Affect the behavior of the DOS emulation environment within a DOS Session. |
| **Memory Extenders** | Affect the behavior of the EMS, XMS, and DPMI memory extenders, if used in the DOS Session. |
| **File Operations** | Affect the behavior of file I/O operations with a DOS Session. |
| **Windowing** | Affect the screen I/O behavior of DOS Sessions when running in windowed mode on the Desktop. |

## Transfer of Data Between DOS Sessions

To allow easy migration of data from DOS applications to PM applications, OS/2 2.0 allows DOS programs to run within windows in the desktop environment. This allows users to display a graph or chart from a DOS application beside a document they are preparing with an OS/2 word processor. A block of text can be selected in one window (a popular DOS word processor, for example) and copied to the Clipboard using a mouse or keyboard. That text can be pasted into another window (perhaps a PM spreadsheet application). Graphic images can also be transferred from a DOS application to the Clipboard.

## Increased Available Memory

The small memory space availability in Version 1.X of the OS/2 prevents some DOS applications from loading. Since the operating system installs some of its kernel, device driver code, and data in fixed memory below 1MB (so system functions can execute in real mode), DOS applications have less space for their own code and data.

**Note:** This problem is not unusual, even in a native DOS environment, since installing memory-resident DOS programs and device drivers have the same effect.

In OS/2 2.0, the available base memory (below 640KB) in each DOS Session is over 620KB, before loading any user-installed DOS device drivers or Terminate-and-Stay-Resident (TSR) programs. This increases the available base memory in OS/2 2.0 by nearly 80KB over Version 1.3. Since the available base memory is greater than in DOS itself, OS/2 users are now able to load some DOS TSR programs with larger DOS applications, which would not fit together in the smaller base memory available in DOS.

Adding LAN drivers to the OS/2 configuration to support the network server or redirector functions does not take up DOS application space. Any user-installed OS/2 device drivers will not affect the amount of application space available to a DOS application running under Enhanced DOS Sessions.

## Memory Extender Support

Many popular DOS applications are now using the Expanded Memory Specification (EMS) or the Extended Memory Specification (XMS) to access memory in protect mode on the 80286 or 80386 processors. This enables DOS applications to access memory above the 1MB real mode addressing limit, to have total code and data space larger than the available base memory, and to have very large code or data objects loaded into memory for enhanced execution speed. The standard configuration of OS/2 2.0 Enhanced DOS Sessions supports both the EMS and XMS functions.

### Expanded Memory Specification

OS/2 2.0 EMS support is based on Lotus**-Intel-Microsoft (LIM) EMS Version 4.0. Under DOS, special hardware is normally required to support EMS, although a number of software-based EMS emulation packages exist. Enhanced DOS Sessions support EMS by mapping memory allocation requests into the linear process address space using normal system memory. No special hardware or software is required. The underlying OS/2 virtual memory management functions provide the expanded memory. Therefore, use of EMS in one DOS Session does not affect the ability of DOS applications in another DOS Session to perform similar EMS functions.

The OS/2 2.0 LIM EMS emulation:

- Implements all the required functions in the LIM EMS Version 4.0.
- Provides each DOS Session with a separate EMS emulation. Each DOS Session has its own set of expanded objects so that features work as they would if each DOS Session was running on a different real 80386. Each DOS Session cannot affect the availability of objects in other DOS Sessions or access memory in other DOS Sessions.
- Provides for remapping of conventional memory (below 640KB) for use by applications.
- Provides configurable limits for how much EMS memory is available across DOS Sessions, as well as a limit per DOS Session. The DOS Settings feature allows the user to override the per DOS Session limit, subject to the constraint given by the overall limit.
- Supports multiple physical-to-single logical mappings. Different 8086 addresses can map to the same expanded memory object address. This is required by some programs.
- EMS can be removed and the operating system can run without loading EMS in any DOS Session.

OS/2 2.0 EMS allows DOS applications to allocate and access up to 32MB of expanded memory in up to 255 EMS objects. These objects can be mapped into the base memory space (below 1MB) so the DOS application may access very large address spaces. Applications access EMS services using the DOS interrupt INT 67h.

EMS services are implemented under Enhanced DOS Sessions using a virtual device driver known as the Virtual Expanded Memory Manager (VEMM). OS/2 2.0 EMS offers a separate EMS emulation to each DOS Session, by placing most VEMM control structures in a per-DOS Session data area outside the V86 mode address space. Unlike most virtual device drivers, VEMM does not have a corresponding physical device driver. Instead, VEMM manages its memory using OS/2 2.0 kernel's memory management services. EMS object allocation, reallocation, or deallocation is accomplished by requesting corresponding services from the operating system's memory manager. For example, when an application requests the allocation of an expanded memory object, VEMM requests the allocation of a memory object in linear memory outside any DOS Session.

VEMM is typically installed at system initialization time, via a DEVICE= statement in CONFIG.SYS, as shown below:

```
DEVICE=C:\OS2\MDOS\VEMM.SYS 4096, 2048
```

*Figure 6-2. Loading VEMM*

To prevent DOS Sessions from using all available memory, there is an overall limit on the amount of EMS memory, and a default limit for each DOS Session to prevent a DOS Session from requesting all available EMS memory. The defaults for these limits are specified in the DEVICE= statement for VEMM. The default limit for each DOS Session can be overridden using the DOS Settings feature.

Setting the overall limit to zero disables EMS in all DOS Sessions, regardless of the per-DOS Session value. Setting the per-DOS Session value to zero disables EMS in all DOS Sessions unless their entries on the desktop specify a nonzero EMS size.

Setting the EMS size to zero for an entry on the desktop disables EMS for that DOS Session. Most users need never change the default value. DOS Settings for frame position, and the size of extra mapping regions above and below 640KB can be configured by the user.

Most VEMM setup is postponed until the first INT 67H service request is made. Only remappable conventional memory is set up before that time. This assures that other virtual device drivers have a chance to reserve read-only memory (ROM) and device memory areas.

## Extended Memory Specification

XMS functions provide another mechanism for DOS applications to access memory above the 1MB limit. OS/2 2.0 Enhanced DOS Session provides support for LIMA Extended Memory Specification Version 2.0, in a similar manner to that provided for LIM EMS Version 4.0, using normal system memory and emulating XMS functions.

The extended memory specification manages three different kinds of memory, as shown in Figure 6-3:

- High Memory Area (HMA)
- Upper Memory Blocks (UMBs) in the Upper Memory Area (UMA)
- Extended Memory Blocks (EMBs)

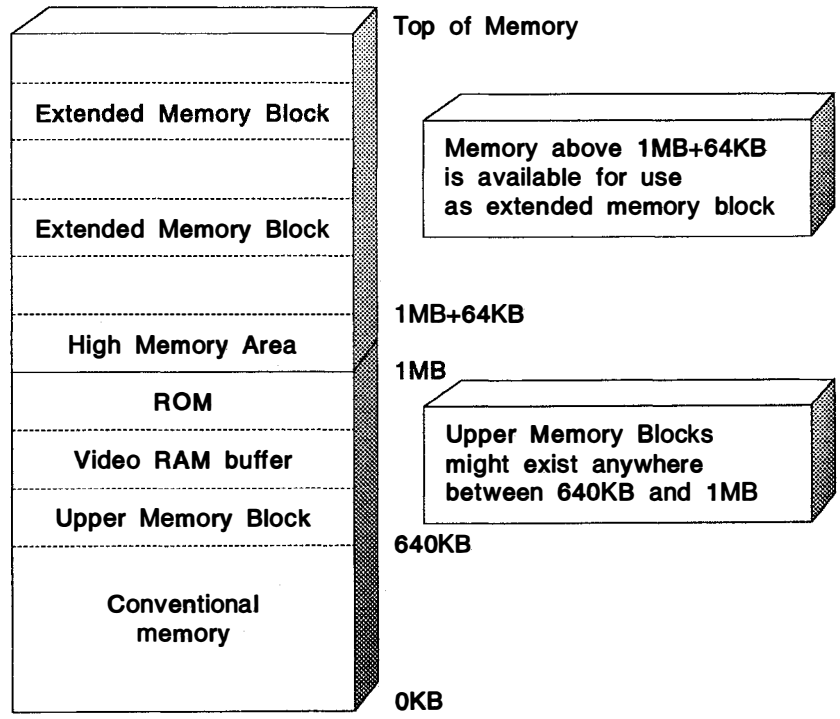



*Figure 6-3. Memory Map of Areas Supported by Extended Memory*

The OS/2 LIMA XMS emulation:

- Implements all LIMA XMS Version 2.0 functions.
- Provides each DOS Session with a separate XMS emulation. Each DOS Session has its own high memory area, upper memory blocks and extended memory blocks; hence programs work as they would if each DOS Session was running on a different real 80386. A DOS Session therefore cannot affect the availability of extended memory objects in other DOS Sessions or access memory owned by other DOS Sessions.
- Provides configurable limits for how much XMS memory is available across all DOS Sessions as well as a limit per-DOS Session. The DOS Settings feature can override the per-DOS Session limit, subject to the constraint given by the overall limit, and can disable XMS altogether for a particular DOS Session if its installation conflicts with the program being run in the DOS Session.
- XMS can be removed and the operating system can run without loading XMS in any DOS Session.

Applications which use extended memory can use the XMS support in the same manner as in a native DOS environment. In addition, portions of the DOS operating system, device drivers and TSR programs can be loaded into extended memory, thereby conserving memory within the DOS application address space for application use.

Note that older applications which access extended memory directly, rather than through an extended memory manager, might not be compatible with the XMS support under Enhanced DOS Sessions.

XMS services are implemented under Enhanced DOS Session using a virtual device driver known as the Virtual Extended Memory Manager (VXMS). VXMS provides a separate XMS emulation for each DOS Session by placing most VXMS control structures in a per-DOS Session data area outside the V86 mode address space.

Like VEMM, and unlike most other virtual device drivers, VXMS does not have a corresponding physical device driver. Instead, VXMS depends on the OS/2 memory manager. XMS object allocation, reallocation and deallocation are accomplished by requesting corresponding services from the operating system's memory manager. For example, when an application requests the allocation of a block of extended memory, VXMS requests the memory manager to allocate a memory object in linear memory outside the V86 mode address space. Reallocation and deallocation are handled similarly.

VXMS is loaded at system initialization time using a DEVICE= statement in CONFIG.SYS, as shown below:

```
DEVICE=C:\OS2\MDOS\VXMS.SYS/XMMLIMIT=8192,2048
```

*Figure 6-4. Loading VXMS*

This statement should be placed in CONFIG.SYS *after* the DEVICE= statement for VEMM.SYS, since VXMS queries VEMM to ensure that no conflicts occur in memory allocation.

The DEVICE= statement uses parameters to specify the total amount of available XMS memory, and the default limit for each DOS Session. In the above example, the overall limit is set to 8MB and the limit for each DOS Session is set to 2MB.

The overall limit comprises the only relationship between XMS memory objects in different DOS Sessions, and is imposed to prevent XMS from acquiring all available memory. The default overall limit is 16MB, and the default limit for each DOS Session is 2MB. The default limit for each DOS Session can be overridden by installing an application on the desktop and choosing to specify the XMS size with the DOS Settings feature.

Setting the overall limit to zero disables XMS in all DOS Sessions regardless of the per-DOS Session value. Setting the default limit for a particular DOS Session to zero disables XMS in all DOS Sessions unless their start list entries specify a non-zero XMS size. Setting the XMS size to zero for an entry in the start list disables XMS for that application's DOS Session only. Most users need never change the default values.

In addition to memory sizes, the number of handles are configurable parameters which may be altered on a per-DOS Session basis using the DOS Settings feature.

XMS supports use of the HIMEM area—a 64KB area just above 1 MB—which can be used for code or data objects by a DOS application. Other XMS functions allow moving code and data objects into extended memory, and from extended memory to base memory. Using these functions, an application can provide several overlays, where one overlay at a time is accessible to the application.

Under Enhanced DOS Sessions, EMS and XMS memory allocations are managed as OS/2 pageable virtual memory, not as fixed physical memory. Therefore, the total memory allocated can exceed the amount of system random-access memory (RAM).

## DOS Protect Mode Interface

Protected mode specifications are such that communication between protect mode and real mode is difficult. A DOS TSR, for example, with which an application communicates through a software interrupt or a shared buffer, will not work in protect mode. The real mode address of the TSR, if used by the protect mode application, will not point to the same memory space as did the same address in real mode because the segment portion of the address is interpreted differently in the two modes.

DOS Protect Mode Interface (DPMI) provides an interface between protect mode and real mode programs. DPMI consists of a set of protect mode functions that, among other things, allocate real mode memory, simulate real mode interrupts and functions calls, and intercept real mode interrupt vectors. DPMI breaks the 640KB memory barrier inherent in DOS application by enabling DOS applications direct access to 32-bit memory segments. OS/2 2.0 Enhanced DOS Sessions provides support for the DPMI Specification Version 0.9, as defined by the DPMI committee.

By using DPMI functions, an application running in protect mode can communicate with the TSR in real mode. The main function that DPMI delivers, however, is that it allows DOS extenders to work properly in a multitasking, protect mode environment.

DOS extenders enable DOS applications to access extended memory while running in protect mode. As required, these extenders switch to:

- protect mode for application code
- real mode for DOS and other real mode calls

DOS extenders under DOS can switch modes on their own. They cannot, however, do so when the application using the DOS extender is running in a virtual machine. The DPMI services provide the same services that DOS extenders use internally to

run their applications. The DOS extender makes INT 31h DPMI calls instead of performing mode switching and selector translation itself.

The term *real mode software* is used to refer to code that runs in the low 1MB address space and uses segment:offset addressing. Under many implementations of DPMI, real mode software is actually executed in virtual 8086 mode. Since virtual 8086 mode closely approximates real mode, V86 mode and real mode are interchangeable in the DPMI context.

The DOS Protect Mode Interface facilitates the following:

- Allows DOS applications to run in protect mode
- Provides DOS applications with access to a large memory address space
- Provides DOS applications with mode switching and calls between V86 ("real" mode) and protect mode
- Provides DOS applications with access to hardware, such as debug registers, in a way that is safe for the host operating system

The DPMI is defined to allow DOS programs to access extended memory while maintaining system protection. DPMI defines a specific subset of DOS and BIOS calls that can be made by protect mode DOS programs. It also defines an interface via software interrupt 31h that protect mode programs can use to allocate memory, modify descriptors and call real mode software. Any operating system that currently supports virtual DOS sessions should be capable of supporting DPMI without affecting system security.

Some DPMI implementations can execute multiple protect mode programs in independent virtual machines. In such implementations, DPMI applications can behave exactly like any other standard DOS program. They can, for example, run in the background or in a window (if the environment supports these features). Programs that run in protect mode gain all the benefits of virtual memory and can run in 32-bit flat model, if desired. OS/2 2.0 provides a DPMI implementation of this nature.

DPMI services are only available to protect mode programs. Programs running real mode can not use these services. Protect mode programs must use a DPMI service to enter protect mode before calling INT 31h services.

Some implementations of DPMI, including OS/2, can run 32-bit 80386 specific programs. The high word of the 32-bit registers is ignored when running 16-bit protect mode programs.

DPMI services are provided by what is referred to as the DPMI *host program*. The programs that use DPMI services are called DPMI *clients*. Generally, DPMI clients fall into two categories:

1. Extended Applications
2. Applications that use DPMI directly

Most DPMI applications are likely to be extended applications. These applications are bound with a DOS extender that is the actual DPMI client. The application calls the DOS extender services. These services are then translated by the client (the DOS extender program) into DPMI calls. The advantage of an extended application over one that calls DPMI services directly is that generally an extender will support more than just DPMI.

A DOS extender can provide a single set of APIs to an application and then translate these APIs to the services (DPMI, EMS, and XMS, for example) that are provided. Where the host extension services are lacking in a particular function the extender must provide that function for the application.

The application code sits on top of a set of base extender functions and APIs. The extender then has separate modules for each type of extension service and code to fill in where services are lacking. An example of a typical extender service is protect mode program loading. The application code that is actually shipped consists of the application code together with the DOS extender code and all of its styles of client support. The host support is generally an extension of the base operating system functions or a device driver used to extend the base operating system functions.

Many programs that use DPMI will be bound to DOS extenders so that they will be able to run under any DOS environment. Existing DOS extenders support APIs that differ from the INT 31h interface. Usually, DOS extenders use an INT 21h multiplex for their extended APIs.

Extenders that support DPMI will need to initialize differently when they are run under DPMI environments. They will need to enter protect mode using the DPMI real to protect mode entry point, install their own API handlers, and then load the DOS extended application program.

DOS extenders should check for the presence of DPMI before attempting to allocate memory or enter protect mode using any other API. When DPMI services are detected, extenders that provide interfaces that extend or are different from the basic DPMI interface will switch into protect mode and initialize any internal data structures. DPMI-compatible extenders that provide no API extensions should simply execute the protect mode application in real mode.

The DPMI implementation in OS/2 2.0 has the following characteristics:

1. DPMI 0.9 is fully supported.
2. DPMI API (INT 31h) support resides outside the kernel and is provided by a virtual device driver (VDPMI.SYS).
3. Both kernel support and the DPMI API layer are independently expandable to future versions of DPMI.

From the application's point of view, it makes no difference whether the actual mode is real mode or V86 mode.

DPMI is service-request driven. An application makes an INT 31h service request. The DPMI VDD handles the request, calling the kernel for basic services, such as allocating memory.

DPMI services are implemented under Enhanced DOS Sessions using the virtual device driver VDPMI.SYS. DPMI is loaded at system initialization time using a DEVICE= statement in CONFIG.SYS, as shown below:

```
DEVICE=C:\OS2\MDOS\VDPMI.SYS
```

*Figure 6-5. Loading DPMI*

Windows 3.0** is not a standard DPMI client and cannot run under DPMI in a DOS Session. Another virtual device driver (VDD), VDPX.SYS, translates system requests (such as INT 21h and BIOS calls) from protect mode to real mode. This VDD will translate the request from a protect mode selector:offset to a real mode address below 1M (segment:offset). Applications can either use this VDD, or provide their own translation services.

Even with DPMI, Windows 3.0 cannot run in Windows Enhanced or Standard Mode under OS/2 2.0. The reason for this is that when Windows runs in Enhanced or Standard Mode it operates at ring 0. Enhanced Mode allows Windows to provide DPMI, virtual memory, hardware virtualization, and multiple DOS boxes for non-Windows applications. Under OS/2 2.0, Enhanced DOS Session performs these functions. When users wish to run Windows applications, they simply start a new DOS Session. (However, the application will run in standard mode.)

## Inside Enhanced DOS Session

The Intel** 80386 Programmer's Reference (1986) describes the system software that supervises Virtual 8086 machines as the V86 Monitor. To provide support for multiple Enhanced DOS Sessions, in OS/2 2.0 and to do this in such a way that would exploit the existing services of the OS/2 kernel (task scheduling, memory management, interprocess communication, and so on), it was necessary to extend the OS/2 kernel itself. The Enhanced DOS Session kernel (Figure 6-1 on page 6-3) is comprised of three new modules, developed using 32-bit code and the OS/2 2.0 flat memory model. These modules—DOS Emulation, 8086 Emulation, and the DOS Session Manager—provide a full set of control program interfaces known as Virtual Device Helper services, which are invoked by virtual device driver (VDD) modules. VDDs provide device-specific support, such as hardware virtualization, BIOS emulation, and other low-level system functions.

The VDD model is of interest to developers whose DOS applications provide hardware-specific functions on non-IBM option adapters. The following section describes the VDD architecture and gives an overview of Enhanced DOS Session operations with emphasis on the behavior of VDDs.

## Virtual Device Helper Services

Several categories of virtual device helper services are available to all VDDs. These services control access to system resources, such as memory, timers, and semaphores. DOS Session events, such as DOS Session foreground and background state switching, can be tracked using virtual device helper services. A VDD can display error messages via the VDHPopUp service. Other virtual device helper services allow a VDD to establish communications with another VDD or with a physical device driver (PDD). Typically, the VDD also must hook a set of system traps that are generated when certain 8086 instructions are executed in V86 mode. To get control, a VDD usually hooks the following traps using the appropriate virtual device helper services:

- Traps generated when a DOS application performs direct I/O to a port that is being virtualized (IN and OUT instructions)
- Traps generated when a BIOS or other software interrupt function is executed in a DOS Session (INT instructions)
- Traps generated when a DOS application attempts to access memory that is mapped to a physical device over which the VDD has control

Another important category of virtual device helper services are those that support the simulation of hardware interrupts. These services control the state of the virtual programmable interrupt controller.

All the virtual device helper service use 32-bit code are callable from assembler language.

For more information on VDH services, see the *OS/2 2.0 Virtual Device Driver Reference*.

## The Virtual Device Driver Model

All OS/2 1.X device drivers are bi-modal; they are required to run in either real mode or protect mode. This is true even if the device driver does not provide explicit DOS mode support (such as hooking a DOS mode software interrupt). Because Enhanced DOS Sessions eliminate real mode execution, existing device drivers should be updated to remove the real mode sections. These drivers are now referred to as Physical Device Drivers (PDDs).

Virtual device drivers (VDD) are new in OS/2 2.0. VDDs are written as 32-bit, flat memory model, protect mode modules. Figure 6-6 shows the structure of virtual device drivers and the interfaces to a PDD.

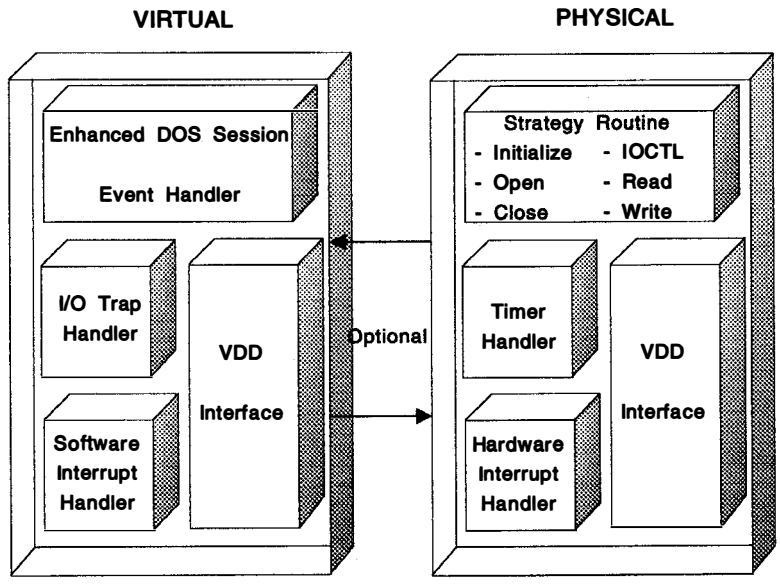



*Figure 6-6. Structure of Virtual and Physical Device Drivers*

All VDDs are loaded and initialized during the system initialization (boot) process. Many of them establish communication at that time with the associated PDD. Table 6-1 on page 6-15 lists the VDDs provided in OS/2 2.0 and indicates those that have a direct interface with an OS/2 PDD. The VDD controls access by any DOS application to the device, and relies on the PDD to manage the physical hardware operations.

*Table 6-1. OS/2 2.0 Virtual Device Drivers*

| VDD | Interfaces with PDD |
|---|---|
| BIOS | |
| CMOS/Real Time Clock | |
| Programmable Interrupt Controller | |
| Timer | X |
| Keyboard | X |
| Mouse | X |
| Disk/Diskette | X |
| DMA Controller | |
| Video (EGA, VGA, and 8514/A) | |
| Printer | X |
| COM | X |
| Expanded Memory Specification | |
| Extended Memory Specification | |
| Numeric Coprocessor (80387) | |
| VCDROM | |
| VDPMI | |
| VDPX | |

For example, the Virtual COM Device Driver has a direct interface to the Physical COM Device Driver. This interface allows the COM PDD to service all the hardware interrupts and to buffer data being transmitted or received. The Virtual COM driver emulates Asynchronous Communications BIOS functions (INT 14H) to send and receive characters and to set or query the state of a COM port. The Virtual COM driver also *virtualizes* the I/O ports associated with a COM port; virtualization implies separate simulation of the physical hardware (in this case, I/O ports) for each DOS Session. This allows the Physical COM driver to manage the actual COM port hardware, while a DOS application accesses only a virtual copy of the port. With this design, an OS/2 application can operate on one COM port while a DOS application accesses a second COM port. Both of these applications can perform data communications in the background. The relationship between the VDD and PDD is shown in Figure 6-7 on page 6-16.

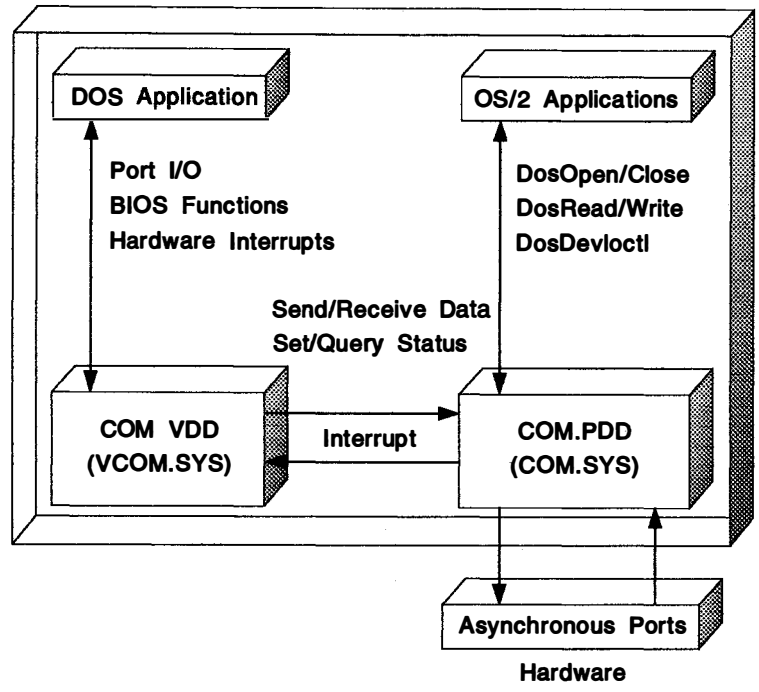



*Figure 6-7. Physical and Virtual Device Drivers Under OS/2 2.0*

Not every VDD needs to operate with a PDD in the same manner as indicated for COM port virtualization, as shown from the list in Table 6-1 on page 6-15. The VDD-PDD interface is required when the device generates hardware interrupts that must be simulated in the context of the DOS Session. For example, many DOS applications that support asynchronous communications include hardware interrupt handler routines. These routines typically perform I/O directly to the COM port hardware rather than going through BIOS.

To support these DOS applications and to allow them to run in the background and foreground, the Virtual COM device driver simulates hardware interrupts in the task-time context of the V86 mode process. DOS Session are scheduled to run using the same preemptive time-slicing task-dispatching method that drives the multitasking OS/2 sessions. Hardware interrupts, on the other hand, occur asynchronously to this task-scheduling process. By simulating the hardware interrupts and presenting a virtual hardware state, the interrupt handling logic of the DOS application does not execute on the physical interrupt thread. This means that switching to V86 mode is not done at interrupt time, but deferred until the scheduler dispatches the DOS Session task.

The advantage of simulated interrupts is that mode switching and hardware virtualization do not need to be done at interrupt time. Also, the DOS application does not get control at interrupt time, which helps to maintain system integrity.

A potential disadvantage of this approach is that DOS applications with routines that are highly real-time dependent might not operate correctly under heavy system load conditions. When developing a VDD, it is important to consider the time-dependent aspects of the DOS application, and to design the hardware virtualization to satisfy this dependency. This can usually be achieved when all hardware states can be simulated in software, and where performance considerations remain high on the VDD-PDD designer's agenda.

## Communication with OS/2 Processes

Another feature of OS/2 2.0 allows protect mode OS/2 processes to request services directly from a VDD. New OS/2 functions, as shown in Table 6-2, support this capability.

*Table 6-2. VDD Services*

| Function | Description |
|---|---|
| DosOpenVDD | Open a Virtual Device Driver (get a handle to a VDD) |
| DosRequestVDD | Allows a protect mode OS/2 application to communicate with a virtual device driver |
| DosCloseVDD | Close a Virtual Device Driver (free a handle to a VDD) |

The VDD designer can provide any appropriate services across this interface, depending on the hardware being virtualized. Potential uses of this interface include the ability of OS/2 applications to communicate with DOS applications. To achieve this, the VDD would also have to provide DOS applications with an interface to establish the communications. This could be done, for example, by defining a series of software interrupt functions that a DOS application could invoke.

## Summary

OS/2 2.0 provides a redesigned DOS compatibility environment known as Enhanced DOS Sessions. Features of Enhanced DOS Sessions include:

- The ability to run DOS applications as V86 mode tasks. This eliminates the operating system overhead of switching between real mode and protect mode, and provides a fully protected system environment.
- The ability to start many concurrent DOS sessions, each operating in its own independent 1MB linear address space.
- The ability to customize the operation of DOS Sessions through DOS Settings.
- The ability to run DOS programs in windows in the PM environment.
- Increased available base memory over previous versions of the operating system.
- Support for Expanded Memory Specification (EMS) and Extended Memory Specification (XMS). This allows DOS applications to access memory above the 1MB real mode addressing limit, to have total code and data space larger than the available base memory, and to have very large code or data objects loaded into memory for enhanced execution speed.

The Enhanced DOS Session is composed of three modules—DOS Emulation, 8086 Emulation, DOS Session Manager—which provide a full set of control program interfaces known as Virtual Device Helper services. These services are invoked by Virtual Device Driver (VDD) modules. VDD modules provide hardware-specific support, such as hardware virtualization, BIOS emulation, and other low-level system functions.

# Chapter 7. Object-Oriented Programming Using SOM

This chapter describes the IBM System Object Model (SOM) for object-oriented programming. It assumes a knowledge of object-oriented programming and design concepts.

## Object-Oriented Programming

Object-oriented programming is a programming paradigm based on objects, which are programming constructs designed to reflect items in the real world. An *object* consists of both the data necessary to describe a real-world item, and the functions necessary to describe the behavior of the item. This is in contrast to the structured programming model, which focuses on the things that can be done to the data (the functions), and which treats the data only as something to be acted on. Objects bind together the data that describes an item and the functions that act on the data.

The basic unit of organization in object-oriented programming is the object, which is a data structure that consists of data and functions. The data is called the *object's state*. The functions that define the object's behavior are called *methods*. Objects are instances, or instantiations, of a class. A *class* is a description of an object. It defines the data that represents the object's state, and the methods that the object supports.

## Object-Oriented Programming Example

An example might make object-oriented programming concepts clearer. A stack is a common programming construct, permitting data to be stored and retrieved in a Last-In, First-Out manner—that is, the last data element placed on the stack is the first element that is retrieved from the stack.

The data structure for the stack describes the stack—a place to store the data put on the stack and a variable to keep track of the location of the top of the stack. Given the definition of the data structure, multiple instances of the stack can be declared within a program.

There are two basic operations that can be performed on a stack, pushing data onto the stack and popping data off of the stack. It would also be beneficial to be able to dynamically create a stack. Functions to perform these activities must be defined.

Figure 7-1 on page 7-2 show the definition of a stack data structure and functions, and the implementation for one of the functions. For clarity, the C programming language is used in this example.

```
/* Define the stack */

struct stackType {
  void *stackArray[STACK_SIZE];
  int stackTop;
};
typedef struct stackType Stack;

/* Define the stack's functions */

Stack *Create();                /* Create a new stack        */
void Push(Stack *thisStack,     /* Push an element onto stack */
          void *nextElement);
void *Pop(Stack *thisStack);    /* Pop an element off stack   */

/* The definition of the Push function is provided as an example.  */
/* The rest of the functions would be defined in a similar manner. */

void Push(Stack *thisStack, void *nextElement)
{
  thisStack->stackArray[thisStack->stackTop] = nextElement;
  thisStack->stackTop++;
}
```

*Figure 7-1. Generic Stack Functions in C*

A client program might use this stack to create a stack of words needing interpretation, as in Figure 7-2.

```
main()
{
  Stack *WordStack;

  char *Subject = "Emily";
  char *Verb = "eats";
  char *Object = "ice cream";
  char *NextWord;

  WordStack = Create();
  Push(WordStack, Object);
  Push(WordStack, Verb);
  Push(WordStack, Subject);

  /* ... */

  while (NextWord = Pop(WordStack)) {
    printf("%s\n", NextWord);

    /* ... */

  }
}
```

*Figure 7-2. Using Generic Stack Functions in C*

The stack is an example of a class. The stack contains two data elements (*stackArray* and *stackTop*), and supports three methods: Create, Push, and Pop. *WordStack* is an object of class Stack; it can also be called an instance of a stack.

Methods must know the specific object on which they are to operate. This object is called the target object, or sometimes the receiving object. Notice that each method (except Create) takes as its first parameter a pointer to the target object. This is because a program might have many objects of a given class, and each are potential targets for the class methods.

## IBM System Object Model

OS/2 2.0 includes a language-neutral object-oriented programming mechanism called the System Object Model (SOM). SOM is specifically designed to support the new, object-oriented paradigm, and to be usable with both procedural (non-object-oriented) languages and object-oriented languages. (This release of SOM only supports the C language.) SOM is not a language—it is a system for defining, manipulating, and releasing class libraries. SOM is used to define classes and methods, while allowing the developer to choose a language for implementing these methods. Most programmers will therefore be able to use SOM quickly without having to learn a new language syntax. SOM consists of a run-time library and a set of utility programs that support building, externalizing, and manipulating software objects.

SOM objects are language-neutral. They can be defined in one programming language and used by applications or objects written in another programming language.

## SOM Features

SOM offers three important object-oriented programming features: encapsulation, inheritance, and polymorphism.

### Encapsulation

An object consists of data and actions (*methods*) that can be performed on that data. An object has an external (*public*) view that prescribes how other objects or applications can interact with it. An object also has an internal (*private*) view that prescribes how data and methods are actually implemented. Object implementation is hidden (*encapsulated*) from the public view.

Developers of SOM objects can externalize as much of an object's definition as they choose. The developers should, however, carefully consider what they choose to externalize. Published methods and instance variables become a permanent part of an object's interface. Unnecessary externalization of an object's definition might compromise future compatibility.

SOM allows changes to an object's internal implementation without affecting the compatibility of resulting binaries. This means that applications using SOM objects will not require recompilation when the SOM object definitions change. Full forward, binary compatibility can be retained when:

- Adding new methods
- Adding, changing, or deleting unpublished instance variables, provided that old methods can still be supported

- Inserting new classes above your class in the inheritance hierarchy
- Relocating methods upward in the class hierarchy

### Inheritance

Inheritance is the derivation of new child classes from existing parent classes. Child classes inherit the characteristics of their parent classes. This means that methods defined for a parent class are automatically defined for the child class. Child classes also can add unique characteristics to their parent classes, in addition to those they have inherited. This means that child classes can define new behavior in terms of new methods. These child classes are also known as subclasses.

### Polymorphism

Polymorphism is, basically, many implementations of the same method for two or more classes of objects. This is known in SOM as method overrides, or override resolution. SOM supports various types of polymorphism, so it can be readily mapped into different object-oriented languages. The following example describes one type of polymorphism: polymorphism by inheritance.

As child classes are derived from parent classes, inherited methods can be overridden. For example, suppose that ClassB is a child of ClassA, as illustrated in Figure 7-3.

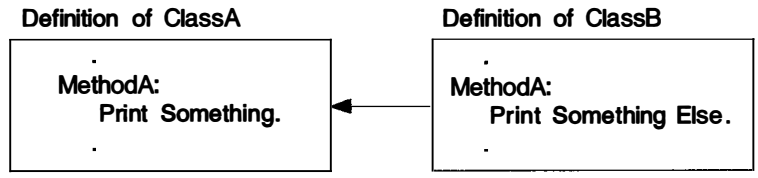



*Figure 7-3. Polymorphism by Inheritance*

MethodA is one of the methods defined for ClassA. It also is one of the methods defined for ClassB, and is inherited from ClassA. In SOM, MethodA can be overridden in the definition of ClassB to do different things for ClassB. MethodA, therefore, is defined for both ClassA and ClassB, but is implemented differently for ClassA and ClassB. This means that method resolution requires the name of the method as well as the object being acted upon.

## The SOM Run-Time Environment

The SOM run-time environment contains the basic data structures and functions that are used to define, create, and manage classes and objects in terms of other classes.

Classes are generic definitions of sets of objects and their behavior. Classes are defined at compilation time. Class objects are the SOM run-time implementation of SOM classes. Because the terms "class" and "class object" refer to the same thing, but in different contexts (compilation time and run time), they can be used interchangeably.

Objects are created dynamically during run time. Objects are instances of classes. The methods that an object responds to are referred to as *instance methods*, because any object instance can perform them. An object's instance methods are defined in its class definition, and cannot be used unless an object instance already exists. Object instances are created by methods that operate on the class object to cause it to produce an object instance. Class methods that create object instances

are called *factory methods*, or constructors. SOM classes that define factory methods for classes are called *metaclasses*. Metaclasses are classes of classes. A class object is an instance of its metaclass.

The relationship between objects, classes, and metaclasses is shown in Figure 7-4.

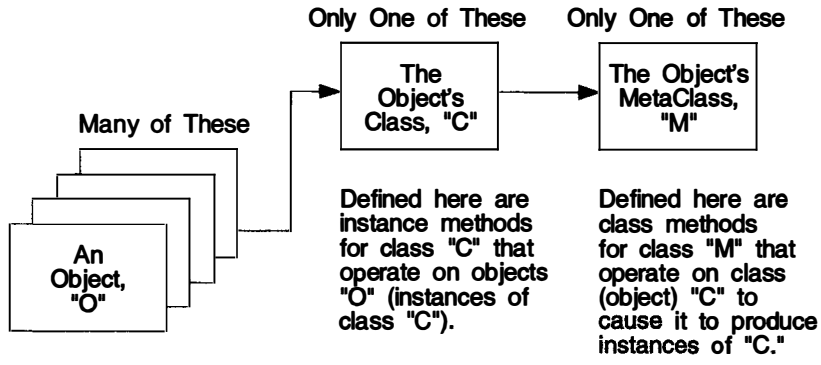



*Figure 7-4. Classes and Metaclasses*

The SOM environment can be created automatically or explicitly within any process that uses it. SOM supplies three classes, as shown in Table 7-1. Classes that make up the SOM run-time environment are packaged with the operating system in SOM.DLL.

*Table 7-1. SOM-Supplied Classes*

| Object | Description |
|---|---|
| SOMObject | Root class for all SOM classes. |
| SOMClass | Root class for all SOM metaclasses. |
| SOMClassMgr | Class for SOMClassMgrObject |

SOMObject defines the essential behavior common to all SOM objects. All SOM classes are subclasses of SOMObject. SOMClass defines the essential behavior common to all SOM class objects. SOMClass is a subclass of SOMObject and is the metaclass of the SOMObject class. By definition, SOMClass is its own metaclass. SOMClassMgr is the class definition for the SOMClassMgrObject that is created during SOM initialization.

During SOM initialization, four objects are created, as shown in Figure 7-5. Three of these objects are class objects.

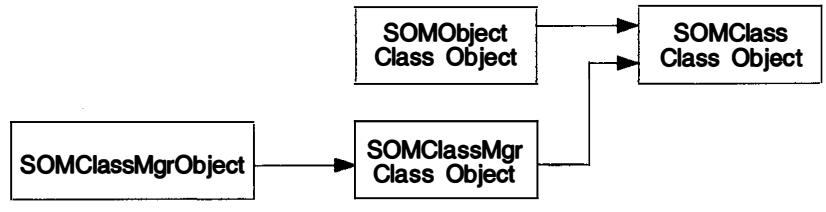



*Figure 7-5. SOM Objects at Initialization*

The SOMClass class object provides constructors for SOMObject class objects and for the SOMClassMgr class object. SOMObject defines a set of methods common to all SOM objects. Because all classes are subclasses of SOMObject, they inherit the set of methods common to all SOM objects. SOMClassMgrObject is an instance of

the SOMClassMgr class object. SOMClassMgrObject dynamically loads and unloads class libraries when referenced and tracks instances of class objects.

## Creating SOM Classes

The first step in creating a SOM class is to define the class and its relationship to other classes. In SOM, a class is defined in a class definition file. Class definition files are ASCII files with an extension of .CSC. The class definition is described in a formal specification language, the SOM Object Interface Definition Language (OIDL). OIDL has a language-neutral core (appropriate for any programming language), to which some minor extensions have been added to simplify programming in C. For OS/2 2.0, OIDL supports the C-programming language only.

The second step is to process the OIDL class definition file using the SOM compiler to produce a set of language-specific or use-specific binding files for the class. The binding files then are used to build class libraries that can be used by client applications; that is, applications that are to create subclasses or object instances.

**Note:** Hereafter, any information concerning SOM bindings is specifically geared toward the SOM C-language bindings, as distinguished from other SOM language bindings.

## Object Interface Definition Language

The class definition file provides a complete description of a class, including its relationship to other classes, its instance data, and the methods that it supports and overrides. An Object Interface Definition Language (OIDL) class definition file for C language also describes information specific to building C-language binding files. An OIDL class definition file for C language is divided into eight sections, as shown in the Class Definition File template in Figure 7-6 on page 7-7. The sections are placed in the template in the recommended order.

```
---Include Section (Required)
---   Directions that tell the SOM compiler where to find the class
---   definitions for the parent class, metaclass, and any ancestor
---   class for which this class overrides one or more of its methods.
---
---Class Section (Required)
---   Name, attributes, and description of the class, and
---   directions that tell the SOM compiler how to name and
---   build the binding files it generates.
---
---Release Order Section (Optional)
---   All method names and public instance variables introduced
---   by the class.
---
---Metaclass Section (Optional)
---   Name and description of metaclass, if not a metaclass of the
---   parent class.
---
---Parent Class Section (Required)
---   Name and description of the parent class.
---
---Passthru Section (Optional)
---   Blocks of code or comments that are not processed by the SOM
---   compiler, but are placed "as is" in the specified binding file.
---
---Data Section (Optional)
---   Instance variables for the objects belonging to this class
---
---Methods Section (Optional)
---   New methods and method overrides supported by the class.
```

*Figure 7-6. Class Definition File Template*

Three of the eight sections are required for a class definition file: the Include, Class, and Parent Class sections. The Include section must describe where the definitions for the class's parent, ancestors, and metaclasses can be found. The minimum information that the Class and Parent Class sections must provide are the names of the class and its parent class.

The Metaclass section is used to specify the name of a class's metaclass. This is only necessary when the class's metaclass is not the same as its parent's metaclass. If this is the case, the location for the metaclass definition must be added to the Include section. If a metaclass name is not specified in this section, then it is assumed (by default) that the class's metaclass is the same as its parent's metaclass.

When a class has the same metaclass as its parent, new methods can be added to the set of class methods, or class methods can be overridden by specifying the CLASS attribute in the Data and Methods sections of the class definition file. When this occurs, SOM creates a subclass of the parent class's metaclass, which then becomes the class's metaclass. The class's metaclass is then referred to as an *implied metaclass*. Creating a subclass of the parent's class can be done explicitly by creating another class definition. But implied metaclasses eliminate the overhead of these additional class definitions.

The Data section of a class definition file describes data elements contained by objects of the class; that is, *instance data*. The Data section contains C-language data declarations for instance data. Instance data can be declared private or public. By default, instance data is private; that is, it can be accessed only by methods of

the class. It cannot be accessed by client applications. Public instance data is part of the published external interface and can be accessed by client applications.

The Methods section of a class definition file describes methods to which objects of this class can respond, including overrides to methods of ancestor classes. The Methods section contains the C-language function prototypes that define the calling sequence for each method new to the class. The Methods sections also contains the names of inherited methods that will be overridden, or implemented differently, by the class. Methods can be private or public. In contrast to instance data, methods are public by default.

The order in which public instance data is declared, or methods are described, in the class definition file is critical for future compatibility, when new public data or methods might be introduced in the class definition. Public instance data can be declared in any order in the Data section of the class definition file. Methods are described in the Methods section of the class definition file, and can be in any order or grouped by function category. By default, when the SOM compiler processes the class definition file, it builds the internal data structures for the binding files based on the order in which the public instance data and methods are described in the class definition file. If, at a later date, new data or methods change the original ordering of public data and methods for the class, the binding files and class library are built differently. Client applications then must be recompiled.

The Release Order section of a class definition file is the SOM solution to maintaining compatibility between changing class definitions and client applications. This section contains a list of all method names and public instance variables introduced by the class. The ordering of this list does not necessarily match the ordering of the instance-data declarations in the Data section or the ordering of the method descriptions in the Methods section. The Release Order list overrides the default processing of the SOM compiler. It directs the SOM compiler to process the data declarations and method definition in the order specified, not as they occur in the class definition file. This means that if a new method is inserted in a method group in the middle of the Methods, section and the method name is added to the end of the Release Order list in the Release Order section, compatibility with client applications can be preserved.

**Note:** If you want the SOM compiler to produce a release order section for you automatically, you must request a .CS2 file. The .CS2 file will have same content as the original class definition file, except that its release order statement will always be complete (even if the release order statement in the class definition file is not), and a consistent comment style will appear throughout.

Figure 7-7 shows the structure of a sample class definition file that groups methods that operate on instance data. In this example, the Release Order list groups related methods and data accordingly; that is, by instance variable and the functions that operate on that instance variable.

```
#include "somobj.sc"

class: example;

release order:  var1, get_var1, set_var1, var2, get_var2, set_var2;

parent:  SOMObject;

data:  var1, public;
       var2, public;

methods:
    group:  get_methods
      get_var1

      get_var2

    group:  set_methods
      set_var1

      set_var2
```

*Figure 7-7. Structure of a Sample Class Definition File*

If methods that operate on a third instance variable are added to the class definition file, compatibility with clients of the Example class defined in Figure 7-7 can be maintained by modifying the release order, as shown in Figure 7-8.

```
#include "somobj.sc"

class: example;

release order:  var1, get_var1, set_var1,
                var2, get_var2, set_var2,
                var3, get_var3, set_var3;

parent:  SOMObject;

data:  var1, public;
       var2, public;
       var3, public;

methods:
    group:  get_methods
      get_var1

      get_var2

      get_var3

    group:  set_methods
      set_var1

      set_var2

      set_var3
```

*Figure 7-8. Maintaining Compatibility by Modifying the Release Order*

In Figure 7-8, the get_var3 and set_var3 methods are inserted in the *get_methods* and *set_methods* groups, respectively. The *var3* instance variable is added to the end of the list of instance data. The new instance data and method names are added to the end of the Release Order list.

Without a Release Order list, the data and methods in the first version of the Example class are processed in the order in which they occur in the file:

`var1, var2, get_var1, get_var2, set_var1, set_var2`

Without a Release Order list, the data and methods in the second version of the Example class are processed in the order in which they occur in the file:

`var1, var2, var3, get_var1, get_var2, get_var3, set_var1, set_var2, set_var3`

Without a Release Order list, the data and method information maintained by SOM in the object data structures built for the first version of the Example class do not match that built for the second version of the Example class. The second version is not compatible with clients of the first version.

A class definition file is made more readable by the use of comments. Several comment styles are supported by OIDL, as shown in Figure 7-9 on page 7-11.

```
/*
 * This is a comment that will appear
 * in the language binding files.
*/

-- This is a comment that will appear
-- in the language binding files.

// This is a comment that will appear
// in the language binding files.

 # This is a comment that will not appear in the
 # language binding files.
```

*Figure 7-9. Syntax of OIDL Comments*

Because OIDL class definition files are used to generate language bindings, comments must be strictly associated with particular elements, so they appear at the appropriate points in the output files. If comments are not placed as prescribed by the OIDL syntax, they might not appear where you expect to see them.

Throw-away comments are not intended to appear in any binding files. They may be placed anywhere in a class definition file. They can be used to comment out sections of a class definition file or add historical or notational commentary.

## Processing Class Definition Files

Figure 7-10 shows how a class definition file is processed by the SOM compiler.

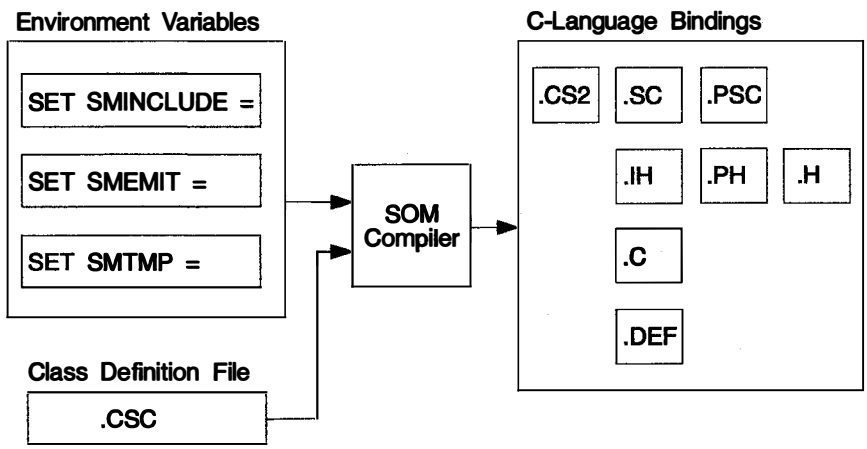



*Figure 7-10. Processing a Class Definition File*

The SOM compiler processes the class definition file for a SOM class and generates a set of language binding files. The file name of a SOM C-language binding file corresponds to that of the class definition file (.CSC) processed by the SOM compiler. Each SOM C-language file has a different extension. For example, the SOM compiler processes EXAMPLE.CSC and generates EXAMPLE.C, EXAMPLE.H, and so forth. The language binding files generated by the SOM compiler are described in Table 7-2.

*Table 7-2. SOM C-Language Bindings Files*

| File Extension | Description |
|---|---|
| .C | Template for C-language source program for the class implementation. |
| .H | Public include file for all C-language programs that need to access the SOM class. |
| .IH | Implementation header containing most of the automatically generated implementation details about the class. |
| .PH | Macros for private methods defined in the class. |
| .DEF | Instructions to the linker about how to build a class library. |
| .SC | Language-neutral form and subset of the SOM class definition file with private-implementation detail removed. This file should be "published" (or exported, made available) to users of the class. |
| .PSC | Supplement to .SC file that contains information about private methods of a class. |
| .CS2 | Stylized form of the original class definition file. |

Some of these files contain the public interface for the class; others contain the private interface. Some files are used to implement the class and its subclasses and some are used by client programs that create and manipulate object instances of the class.

**Note:** The SOM compiler can also accept .SC files as input. In this case, only the .H files that are created are meaningful, but this is a major use of the SOM compiler. In those cases where only the .SC files are published, these files can be used to generate .H files. In order to obtain a neutral .H file for use with any C compiler, compile the .SC file as shown in Figure 7-11.

```
SC -s "h" -a cl386 xyz.sc
```

*Figure 7-11. (SOM) Compiling .SC Files*

The .IH and .C files are the C-language source files for the class implementation. The .IH file is automatically included in the .C file. The .SC file is specified in the Include section of the class definition files for subclasses of the class. The .H file must be included in client programs to create and manipulate object instances of the class. The .PH and .PSC files are the counterparts to the .IH and .SC files. These contain the private interface for the class and should be reserved only for class implementers who need to access the class's private methods.

A set of environment variables, as shown in Table 7-3, control SOM-compiler processing. SMTMP is optional and defaults to the root directory of the current drive. If the files specified in the Include section of the class definition file are enclosed in double quotation marks, SMINCLUDE is optional and defaults to the root directory of the current drive. If the files specified in the Include section of the class definition file are enclosed in angled brackets (< >), SMINCLUDE is required for SOM-compiler processing. SMEMIT is used to indicate which bindings files are generated.

| Table 7-3. SOM Environment Variables | |
|---|---|
| **Variable** | **Description** |
| SMINCLUDE | Specify location of class definitions. |
| SMEMIT | Specify which binding files are to be generated. |
| SMTMP | Specify directory SOM can use for intermediate files. |

## A Simple Class Implementation

The language binding files generated by the SOM compiler include a template for the C-language source program for the class implementation. This program template contains stub procedures for all new and override methods specified in the class definition file. The application developer must supply the code implementation for the stub-method procedures. Figure 7-12 through Figure 7-18 on page 7-15 illustrate the stages in this process.

```
--- Dog.CSC
---
--- A class defining a set of objects (dogs)

#include "somobj.sc"

class:  Dog;

parent: SOMObject;

methods:

   void bark();
   ---Have the dog bark.
```

*Figure 7-12. Class Definition File*

In Figure 7-12, the Dog class is defined. According to this class definition file, the Dog class is derived from the SOMObject class. The behavior of the SOMObject class is defined in the file, SOMOBJ.SC, which was generated by the SOM compiler when the SOMObject class was implemented. Object instances of the Dog class inherit the behavior of its parent class SOMObject by specifying the SOMOBJ.SC file in the Include section of the Dog class definition file. This means that all methods that can act on instances of SOMObject can act on Dog class objects.

In the real world, dogs have many characteristics and behaviors. A comprehensive definition of the Dog class would include methods that relate to these characteristics and behaviors. For this example, the only behavior that is defined is "barking". The behavior, "dogs can bark" corresponds to the prototype of the "bark" method specified in the Methods section of the class definition file.

```
#define Dog_Class_Source

#include "dog.ih"

SOM_Scope void SOMLINK bark(Dog *somSelf)
  {
   DogData *somThis = DogGetData(somSelf);
   DogMethodDebug("Dog", "Bark");
  }
```

*Figure 7-13. C-Language Source-Program Template for Implementation of Dog Class*

The SOM compiler processes the DOG.CSC file as shown in Figure 7-12 on page 7-13 and generates the DOG.C C-language source-program template shown in Figure 7-13. Notice that DOG.IH, another file generated by the SOM compiler, is automatically included in this program template. This file contains the data structures, macros, and functions for accessing data and methods for object instances of the Dog class. This file also provides stubs for all methods prototyped in the Methods section of the DOG.CSC file. The general form for a method stub is shown in Figure 7-14.

```
SOM_Scope void SOMLINK <methodname>(<classname> *somSelf)
  {
   <classname>Data *somThis = <classname>GetData(somSelf);
   <classname>MethodDebug("<classname>", "<methodname>");
  }
```

*Figure 7-14. General Form for a Method Stub*

SOM_Scope and SOMLINK are C macros used internally with the C-binding files.

*somSelf* is a pointer to an object instance of the class <classname>. Because the same method can be invoked on different objects but implemented differently (polymorphism), a pointer to the object being operated on is required as a parameter in the method invocation. The first parameter in the method invocation is always *somSelf*. This means that if the method prototype in DOG.CSC for the "bark" method is specified as shown in Figure 7-15, the SOM compiler generates the C-language source-program template file as shown in Figure 7-16.

```
void bark(int x);
```

*Figure 7-15. Prototype of Method with Parameters*

```
SOM_Scope void SOMLINK bark(Dog *somSelf, int x)
  {
   DogData *somThis = DogGetData(somSelf);
   DogMethodDebug("Dog", "Bark");
  }
```

*Figure 7-16. Stub of Method with Parameters*

The pointer *somThis* is a pointer to data for an object instance of the class <classname>. <classname>Data is a class data type automatically generated by the SOM compiler and placed in the .IH file.

<classname>GetData and <classname>MethodDebug are class macros automatically generated by the SOM compiler and placed in the .IH file. <classname>GetData gets the data for the object instance (somSelf) of the class.

**Note:** Methods that access the object's data use the <classname>GetData macro to establish addressability. This macro must be one of the first executable lines of code in each method, and the value it returns should be assigned to a local variable named "somThis." The SOM compiler automatically generates the code that accomplishes this in each method stub in a .C file.

<classname>MethodDebug provides method-tracing capabilities. This custom macro is generated as part of the method stubs produced in the .C program template. It takes two arguments—a class name and a method name—and if SOM_TraceLevel has the value 1 or 2, produces a message each time a method is entered. (Setting SOM_TraceLevel to 2 also causes the methods supplied as part of the SOM run time to generate method trace output.) To suppress the generation of method tracing code, place a line similar to the one shown in Figure 7-17 in your .C file after the #include statement for <classname>.IH:

```
#define <classname>MethodDebug(c,m) SOM_NoTrace(c,m)
```

*Figure 7-17. Suppressing SOM Tracing*

To complete the class-implementation process, the application developer modifies the C-language source-program template as shown in Figure 7-18. The developer must supply the code for each of the stubbed method procedures. In the example, "dog barking" is implemented as "printing the sound a dog makes," or, "printing Unknown Dog Noise." Because the C-library PRINTF routine is used to implement this function, STDIO.H also must be included in the source program.

```
#define Dog_Class_Source

#include "dog.ih"
#include <stdio.h>

SOM_Scope void SOMLINK bark(Dog *somSelf)
  {
    DogData *somThis = DogGetData(somSelf);
    DogMethodDebug("Dog", "Bark");
    printf("Unknown Dog Noise\n");
  }
```

*Figure 7-18. C-language Source Program for Implementation of Dog Class*

## SOM Macros, Functions, and Data

In order to effectively use the SOM C-language bindings, an understanding of the SOM naming conventions is needed. These are summarized in Table 7-4.

*Table 7-4. SOM Naming Conventions*

| Prefix | Use |
|---|---|
| som | Function and Method Names |
| SOM | Data items |

For example, methods for the SOM objects provided by the SOM run time have the prefix, "som." Constants, data types, and pointers to functions (this includes macro names) have the prefix, "SOM."

Next, and most importantly, you need to understand SOM macros. SOM macros are used by class implementers and by client programs to:

- Instantiate objects
- Access object variables
- Invoke object methods

SOM macros shield the programmer from the details and complexity of SOM data structures, method resolution, and function invocations. There are two types of SOM macros:

- Class-specific
- Non-class-specific, or general

### Class-Specific SOM Macros

Class-specific macros resolve references to class methods, class functions, and class instance data. The macro name or macro parameters contain a class method name, a class function name, an instance variable name, or the class name. The SOM compiler automatically generates them and places them in the class header files. Class implementers can use them by including the class-implementation header file (.IH) in their source programs. Client programs can use them by including the public class header file (.H). Because the .IH file includes the .H file, some class macros (those defined in the .H file) are available to both class implementers and clients, and some (those defined in the .IH file but not in the .H file) are available only to class implementers.

The simplest macro is the "_" (underscore) macro. Object instance data can be referred to by preceding the name of the data element with an underscore character, as shown in Figure 7-19. Underscored-data-name macros are defined in the .H public class header file and are available to both class implementers and client programs.

```
return _var1;
```

*Figure 7-19. Underscored-Data-Name macro*

Object methods can be similarly referred to and invoked by preceding the method name with an underscore character, as shown in Figure 7-20 on page 7-17. Underscored-method-name macros also are defined in the .H public class header file and are available to both class implementers and client programs.

```
_methodA(obj, x, y)
/* method A, with parameters x and y, operates on object obj */
```

*Figure 7-20. Underscored-Method-Name macro*

When nonrelated classes independently define methods with the same name, their methods can be invoked with a variation of this macro. The method name is prefixed with the underscore character and class name, as shown in Figure 7-21.

```
#include "classa.h"
#undef _methodA
#include "classa.h"

ClassA_methodA (obj1); /* methodA, defined for ClassA objects,          */
                       /* operates on ClassA instance object obj1       */

ClassB_methodA (obj2); /* methodA, defined for unrelated ClassB objects,*/
                       /* operates on ClassB instance object obj2       */
```

*Figure 7-21. Invoking Methods with Identical Names but Nonrelated Classes*

Class objects also can be referred to by preceding the class name with an underscore character. Underscored-class-name macros also are defined in the .H public class header file and are available to both class implementers and client programs.

Other class-specific macros are summarized in Table 7-5.

*Table 7-5. Class-Specific Macros*

| Function | Macros |
|---|---|
| Instantiate objects | <classname>New<br><classname>Renew |
| Access instance data | get_<instance variable>,<br><classname>GetData |
| Invoke methods | SOM_Resolve<br>SOM_ResolveNoCheck |
| Invoke parent methods | parent_<methodname><br>SOM_ParentResolve |
| Trace methods | <classname>MethodDebug |

Of the macros listed in Table 7-5, <classname>GetData and the parent method macros are defined in the .IH file, but not in the .H file. They are available only for class implementers.

A class-specific function also is defined in the .H file associated with a class. This function, <classname>NewClass, creates the class object. It is invoked automatically when an object is instantiated through the <classname>New class macro.

## General SOM Macros and Functions

SOM provides a set of non-class-specific, or general SOM macros and functions that support:

- ID manipulation
- Debugging
- Error handling
- Getting object information

General SOM macros are defined in the SOM.H header file. Because SOM.H is included in the class header (.H and .IH) files, SOM macros are available to class implementers and client programs.

***SOM ID Manipulation:*** IDs are numbers that uniquely represent strings. They can be used in SOM to identify method names, class names, and descriptors. Typically, they are used to provide a fast and efficient means of comparing the strings they represent.

SOM provides a set of macros and functions, as shown in Table 7-6, that can be used to manipulate SOM IDs.

*Table 7-6. Macros and Functions for Manipulating SOM IDs*

| Type | Interface |
|---|---|
| Macros | SOM_CheckID<br>SOM_CompareIDs<br>SOM_StringFromID<br>SOM_IDFromString |
| Functions | somRegisterId<br>somUniqueKey<br>somTotalRegIds<br>somSetExpectedIds<br>somBeginPersistentIds<br>somEndPersistentIds |

Initially an ID is a pointer to a string. A SOM ID is automatically converted to an internal ID representation by the SOM_CheckID macro or by the first invocation of any of the ID manipulation macros. Because the representation of an ID changes, SOM IDs are of a special data type (*somId*).

***SOM Debugging:*** The SOM run-time library provides a means of generating character output by invoking macros and functions that call a replaceable SOM procedure called SOMOutCharRoutine. These macros and functions are available to assist the application developer with debugging an application. Output generated by the debug macros can be conditionally suppressed or produced based on the setting of four global variables. Table 7-7 summarizes the SOM debugging macros and the global variables that affect them.

*Table 7-7. SOM Debug Macros and Control Variables*

| Macro | Control Variables |
|---|---|
| SOM_TestC | SOM_WarnLevel |
| SOM_WarnMsg | SOM_WarnLevel |
| SOM_Assert | SOM_AssertLevel |
| SOM_Expect | SOM_WarnLevel |

In addition to the debug macros, SOM provides a function, somPrintf, that unconditionally generates character output. The interface to somPrintf is identical to the printf C-library routine.

***SOM Error Handling:*** The SOM run-time library also provides a way to handle SOM errors by invoking macros (SOM_ERROR and SOM_TEST) that call a replaceable SOM procedure called SomError. SomError produces a message, an error code, and can, if appropriate, end the process where the error occurred. SOM errors are classified by severity, which is indicated in the low-order digit of the SOM error code. There are three SOM error classes, or severity levels, as shown in Table 7-8.

*Table 7-8. SOM Error Severity Levels*

| Severity Level | Description |
|---|---|
| SOM_Ignore | Normal and informational only |
| SOM_Warn | Abnormal but not unrecoverable |
| SOM_Fatal | Abnormal and unrecoverable |

***Getting SOM Object Information:*** A SOM class implementer or client program can easily determine the class of an object by invoking the SOM_GetClass macro. This macro returns a pointer to the class object. All SOM class objects support methods (for example, somGetInstanceSize) that return information about the objects they create. Therefore, by determining the class of an object and invoking class object methods, more can be learned about the original object.

***Replaceable SOM Functions:*** The SOM run-time environment uses SOM functions that perform memory management, DLL management, character output, and error handling. These functions are replaceable. This means that you can override them by supplying your own version of the default SOM functions. Replaceable SOM functions are summarized in Table 7-9.

*Table 7-9. Replaceable SOM Functions*

| Category | Functions |
|---|---|
| Memory Management | SOMCalloc<br>SOMFree<br>SOMMalloc<br>SOMRealloc |
| DLL Management | SOMClassInitFuncName<br>SOMDeleteModule<br>SOMLoadModule |
| Character Output | SOMOutCharRoutine |
| Error Handling | SOMError |

Figure 7-22 on page 7-20 shows how a user-defined function can be substituted for one of the replaceable SOM functions.

```
#include <som.h>
/* Define your replacement routine                                    */
int myReplacementForSOMOutCharRoutine (char c)
{
     (Your code goes here.)
}
.
.
.
/* After the next statement all output will be sent to your routine */
SOMOutCharRoutine = myReplacementForSOMOutCharRoutine;
```

*Figure 7-22. Replacing SOM Functions*

## Invoking Methods and Accessing Data

The basic rules for invoking methods and accessing object data can be summarized as follows:

- Object data can be referred to by preceding the name of the data element with an underscore character. This is only valid in a method of the class, and then only for the object being operated on.
- Methods can be invoked by preceding the method name with an underscore character. Method parameters always include a pointer to the object being operated on. Invocation of instance methods requires a pointer to an instance object. Invocation of class methods requires a pointer to a class object.

These rules can be illustrated by introducing three new methods for the Dog class that relate to more dog characteristics and getting and setting behaviors. Because dogs can be characterized by their breed, a user should be able to get and set the breed for a dog. It would also be desirable to view or display the characteristics of a dog. In Figure 7-23 on page 7-21, the action of getting and setting a dog's breed corresponds to the prototype for the getBreed and setBreed methods in the DOG.CSC file. The action of displaying a dog's characteristics corresponds to the prototype for the display method.

```
--- Dog.CSC
---
--- A class defining a set of objects (dogs)

#include "somobj.sc"

class:  Dog;

parent: SOMObject;

data:
   char breed[100]

methods:
   void setBreed(char *myBreed);
   ---Set the breed of this dog.

   char *getBreed();
   ---Get the breed of this dog.

   void display();
   ---Display characteristics of this dog.
   ---Display its breed and have it bark.

   void bark();
   ---Have the dog bark.
```

*Figure 7-23. Invoking Methods and Accessing Data in a New Dog Class (DOG.CSC)*

Figure 7-23 also introduces the use of instance data that supports the methods that can operate on dog objects. Each dog has its own "copy" of the instance data associated with it. In this example, the breed of a dog is stored in a string called "breed". The setBreed method fills the string with the appropriate characters. The getBreed method gets the contents of the string.

From the new DOG.CSC file, the SOM compiler generates a new C-language source-program template, which is used to complete the implementation of the new Dog class, as shown in Figure 7-24 on page 7-22.

```
#define Dog_Class_Source

#include "dog.ih"
#include <string.h>
#include <stdio.h>

SOM_Scope char *SOMLINK setBreed(Dog *somSelf, char *myBreed)
   {
    DogData *somThis = DogGetData(somSelf);
    DogMethodDebug("Dog", "setBreed");
    strcpy(_breed, myBreed);
   }

SOM_Scope char *SOMLINK getBreed(Dog *somSelf)
   {
    DogData *somThis = DogGetData(somSelf);
    DogMethodDebug("Dog", "getBreed");
    return _breed;
   }

SOM_Scope void SOMLINK display(Dog *somSelf)
   {
    DogData *somThis = DogGetData(somSelf);
    DogMethodDebug("Dog", "display");
    printf ("\nMy breed is %s\n,
            _getBreed(somSelf));
    printf ("I say\n");
    _bark(somSelf);
   }

SOM_Scope void SOMLINK bark(Dog *somSelf)
   {
    DogData *somThis = DogGetData(somSelf);
    DogMethodDebug("Dog", "Bark");
    printf ("Unknown Dog Noise\n");
   }
```

*Figure 7-24. Implementation of a New Dog Class (DOG.C)*

The first thing that is noticeable in Figure 7-24 is that the instance-data declarations are not present. They have been placed in the class-implementation file (DOG.IH) by the SOM compiler. As before, DOG.IH contains all the definitions of the Dog class data, macros, and functions.

The setBreed and getBreed methods operate on the string, "breed," and refer to it by prefixing the string name with the underscore character. The setBreed method uses the C-library routine, strcpy, to fill the string with the dog's breed, as specified as a parameter to the method. To resolve the reference to strcpy, STRING.H is included in the source program.

The display method requires that the dog's characteristics (that is, its breed and its bark) are displayed. To do this, the display method calls both the getBreed and bark methods, which are referenced by prefixing the method name with the underscore character.

Finally, as in the previous implementation of the Dog class, the include file, STDIO.H, is added to the source program to resolve the reference to the C-library routine, printf, which is used to display the dog's characteristics. The SOM function,

somPrintf, can be used in place of printf; no additional include file is required in this case. In fact, it is recommended that SOM class implementations use the SOM functions, where appropriate, instead of the C-library functions. SOM functions offer flexibility and replaceability.

## A SOM Client Program

So far, the examples have shown how to define and implement SOM classes, but have only mentioned SOM client programs. SOM client programs are applications that create and manipulate SOM objects.

A SOM client of the new Dog class can, for simplicity, create an instance of the Dog class, define its breed, and display its characteristics. This simple SOM client is shown in Figure 7-25.

```
#include "dog.h"

int main()
{
   Dog *Zack;
   Zack = DogNew();
   _setBreed(Zack, "Yorkshire Terrier");
   _display(Zack);
   _somFree(Zack);
   return 0;
}
```

*Figure 7-25. Client of the New Dog Class*

To create and manipulate SOM objects, a client program must have access to the object's public methods. In the same way that class data, methods, and functions are available to class implementers through the .IH file associated with the class implementation, class data methods and functions are available to client programs through the .H file associated with the class implementation. In the client program example in Figure 7-25, DOG.H is included to resolve references to the dog object's public methods.

In the example, the variable *Zack* is defined as a pointer to an instance (object) of the Dog class. In general, a pointer to an instance of a class is declared as shown in Figure 7-26.

```
<classname> *object;
```

*Figure 7-26. Defining an Object*

The DogNew Dog class macro then is used to create an instance of the Dog class and return the pointer in the variable *Zack*. DogNew is defined in DOG.H and is a method inherited from the parent (SOMObject) of the Dog class and tailored for the Dog class. The DogNew macro expands to invoke the somNew method. somNew invokes the DogNewClass function, which creates the Dog class object, if it has not yet been created. The Dog class object must be created before its instances can be created. If instances of the Dog class are created through some mechanism other than DogNew, the DogNewClass function must be invoked in the client program.

Because the setBreed and display methods are public and are defined in DOG.H, the client program can invoke them in the same manner as the class

implementation: by prefixing the method name with the underscore character. In the client program, the pointer to the dog object (Zack) is the first parameter for these methods. The setBreed method is called to set Zack's breed as "Yorkshire Terrier." The display method invokes the getBreed and bark methods and prints the dog's breed and bark.

Finally, the somFree method releases resources allocated when an object is created by somNew. As previously mentioned, somNew is invoked by the DogNew macro. The somFree method, like somNew, is a method inherited from the parent (SOMObject) of the Dog class. The somFree method must be called if somNew is used to create an object.

The output from the client program is shown in Figure 7-27.

```
My breed is Yorkshire Terrier
I say
Unknown Dog Noise
```

*Figure 7-27. Output from Client of Dog Class*

## Inheritance and Polymorphism: Overriding Methods

In the Dog class example, the Dog class is derived from SOMObject. Object instances of the Dog class inherit SOMObject behavior; that is, all SOMObject methods can operate on instances of the Dog class. In addition, the Dog class example defined methods not defined for SOMObject. The Dog class is a subclass of SOMObject.

**Note:** All classes are derived, either directly or indirectly, from SOMObject.

LittleDogs and BigDogs, subclasses of the Dog class, can be defined. These subclasses inherit the behavior of their parent class (Dog class), as well as the behavior of their parent's parent class (SOMObject). If the Dog class had been derived from other classes that were derived from SOMObject, the new subclasses would also inherit the behavior of these ancestor classes. In addition to adding new methods to those inherited from ancestor classes, subclasses can modify or override any inherited methods.

The inheritance relationship between the new subclasses (LittleDog and BigDog) and their ancestors (Dog and SOMObject) is shown in Figure 7-28.


*Figure 7-28. Inheritance Relationships Between Classes and Subclasses*

LittleDogs and BigDogs can be differentiated by the sound, or bark, they make. For this example, instead of making an "Unknown dog noise," little dogs "Yap" and big dogs "WOOF." This means that the LittleDog and BigDog classes implement the bark method inherited from the Dog class differently. In the class definition files for the LittleDog and BigDog classes (Figure 7-29 and Figure 7-30), this is indicated as a method override.

```
include "dog.sc"

class:  LittleDog;

parent:  Dog;

methods:
   override bark;
```

*Figure 7-29. LDOG.CSC—LittleDog Class Definition File*

```
include "dog.sc"

class:  BigDog;

parent:  Dog;

methods:
   override bark;
```

*Figure 7-30. BDOG.CSC—BigDog Class Definition File*

Because the parent of the LittleDog and BigDog classes is the Dog class, DOG.SC must be included in the class definition files so that methods of the Dog class are inherited and can be referenced.

The implementations of the LittleDog and BigDog classes are similar to the implementation of the Dog class. In the implementation of the LittleDog class, as shown in Figure 7-31, the include file, LDOG.IH, contains the definitions for LittleDog class data, macros, and functions. The implementation of the bark method reflects the yapping of little dogs.

In the implementation of the BigDog class, as shown in Figure 7-32, the include file, BDOG.IH, contains the definitions for BigDog class data, macros, and functions. The implementation of the bark method reflects the WOOFing of big dogs.

```
#define LittleDog_Class_Source
#include "ldog.ih"
#include <stdio.h>

SOM_Scope void SOMLINK bark(LittleDog *somSelf)
{
  LittleDogData *somThis = LittleDogGetData(somSelf);
  LittleDogMethodDebug("LittleDog","bark");
  printf("yap yap\n");
  printf("yap yap\n");
}
```

*Figure 7-31. LDOG.C—LittleDog Class Implementation*

```
#define BigDog_Class_Source
#include "bdog.ih"
#include <stdio.h>

SOM_Scope void SOMLINK bark(BigDog *somSelf)
{
  BigDogData *somThis = BigDogGetData(somSelf);
  BigDogMethodDebug("BigDog","bark");
  printf("WOOF WOOF\n");
  printf("WOOF WOOF\n");
  printf("WOOF WOOF\n");
  printf("WOOF WOOF\n");
}
```

*Figure 7-32. BDOG.C—BigDog Class Implementation*

In order for a client program of the Dog, BigDog, and LittleDog classes to access public methods for these classes, the client program must include the public class header (.H) files for the respective classes. In Figure 7-33 on page 7-27, the client of these classes includes DOG.H, LDOG.H, and BDOG.H files. Figure 7-34 on page 7-27 shows the output from this client program.

```
#include "dog.h"
#include "ldog.h"
#include "bdog.h"

int main()
{
   Dog *Pokey;
   LittleDog *Zack;
   BigDog *Pepper;

/* Set pointers to instantiated objects
   ------------------------------------*/
   Pokey = DogNew();
   Zack = LittleDogNew();
   Pepper = BigDogNew();

/* Set their breeds
   ------------------------------------*/
   _setBreed(Pokey, "Basset Hound");
   _setBreed(Zack, "Yorkshire Terrier");
   _setBreed(Pepper, "Rottweiler");

/* Display dog characteristics
   ------------------------------------*/
   _display(Pokey);
   _display(Zack);
   _display(Pepper);

/* Free objects
   ------------------------------------*/
   _somFree(Pokey);
   _somFree(Zack);
   _somFree(Pepper);
   return 0;
}
```

*Figure 7-33. Client of Dog, BigDog, and LittleDog Classes*

```
My breed is Basset Hound
I say
Unknown Dog Noise

My breed is Yorkshire Terrier
I say
yap yap
yap yap

My breed is Rottweiler
I say
WOOF WOOF
WOOF WOOF
WOOF WOOF
WOOF WOOF
```

*Figure 7-34. Output from Client of Dog, BigDog, and LittleDog Classes*

## Metaclasses

In the implementation of the Dog, BigDog, and LittleDog classes, a metaclass is not specified in the Metaclass section of the class definition files. This means that the metaclass of each of these classes is the metaclass of their parent class. The metaclass of the BigDog and LittleDog classes is the metaclass of its parent class (Dog). The metaclass of the Dog class is the metaclass of its parent class (SOMObject), and the metaclass of the SOMObject class is SOMClass. SOMClass, then, supplies the constructors for the Dog, BigDog, and LittleDog classes, as well as for SOMObject.

If a metaclass is specified in the Metaclass section of the class definition files, a new metaclass is defined for the classes. The new metaclass provides new constructors for the classes. As an example, if a new metaclass, DogMeta, is defined for the Dog class, the class definition file for this class is shown in Figure 7-35.

```
include <somcls.sc>

class:  DogMeta;

parent:  SOMClass;

methods:
  SOMAny *newDog();
  --Create an instance of the dog class
```

*Figure 7-35. DOGMETA.CSC—Class Definition for Metaclass of Dog Class*

Because SOMClass is the root class for all metaclasses, DogMeta is derived from SOMClass. This is reflected in the Parent and Include sections in the class definition file for the DogMeta class. The only method that DogMeta defines is newDog, the constructor that creates an instance of the dog class.

```
#define DogMeta_Class_Source
#include "dogmeta.ih"
#include "dog.h"

SOM_SCOPE SOMAny *SOMLINK newDog(DogMeta * somSelf)
   {
    DogMetaData *somThis = DogMetaGetData(somSelf);
    DogMetaMethodDebug("DogMeta", "newDog");
    Dog *aDog;
    aDog=_somNew(somSelf);
    return(aDog);
   }
```

*Figure 7-36. DOGMETA.C—Implementation of DogMeta Class*

The implementation of the DogMeta class, as shown in Figure 7-36, is similar to the implementation of the Dog class. However, DOGMETA.C is also a client of the Dog class: the newDog method returns a pointer to an instance of the Dog class. To resolve this data type during the C-language compilation of DOGMETA.C, DOG.H must be included.

Before the C-language compilation of DOGMETA.C is done, the Dog class definition file must be redefined to associate the new metaclass with the Dog class. The new class definition file for the Dog Class is shown in Figure 7-37 on page 7-29.

```
--- Dog.CSC
---
--- A class defining a set of objects (dogs)

include "dogmeta.sc"
include "somobj.sc"

class:  Dog;

metaclass:  DogMeta;

parent: SOMObject;

methods:

   void bark();
   ---Have the dog bark.
```

*Figure 7-37. DOG.CSC—Associating a Metaclass with a Class*

The new Dog class definition file has a Metaclass section. The .SC file associated with the DogMeta metaclass is now required in the Include section. When the SOM compiler processes DOG.CSC, it generates a new DOG.H file that includes DOGMETA.H.

The final implementation of the new Dog class is identical to the previous example. Clients of the Dog class, however, instantiate instances of the Dog class differently, as shown in Figure 7-38.

```
#include "dog.h"

int main()
{
   DogNewClass(Dog_MajorVersion, Dog_MinorVersion);

   Dog *Zack;

   Zack = _newDog();
   _setBreed(Zack, "Yorkshire Terrier");
   _display(Zack);
   _somFree(Zack);
   return 0;
}
```

*Figure 7-38. Client of Dog and DogMeta Classes*

The constructor method (that is, the method that creates instances of an object) for the Dog class is now newDog. In the previous examples, the DogNew macro was used to instantiate Dog objects. Because the DogNew macro invokes the DogNewClass function to create the Dog class object, the previous client programs did not have to invoke DogNewClass directly. Because the implementation of the

newDog method does not call the DogNewClass function, the new client program is required to do so.

The parameters for the DogNewClass function are defined by the class implementer in the DOG.CSC file. MajorVersion and MinorVersion are attributes defined in the Class section of the class definition file and used by the DogNewClass function to determine compatibility with versions of the class library. In this example, Dog_MajorVersion and Dog_MinorVersion have not been previously defined in the CLASS section of the DOG.CSC file.

## Implied Metaclasses

If a metaclass is not specified in the Metaclass section of a class definition file, the class's metaclass is, by default, its parent's metaclass. This was demonstrated in the earliest Dog class examples. If a class's metaclass is different from its parent's metaclass, then it must be specified in the Metaclass section of the class definition file. This was demonstrated in the latest Dog class example. The same process can be followed to derive a class's metaclass from its parent (that is, create a subclass of the parent's metaclass). This requires a separate .CSC file and generates associated binding files for each class and metaclass.

A subclass of a parent's metaclass can also be created by using implied metaclasses. This is accomplished by defining new and overriding existing class methods within the class's .CSC file.

Using the earliest Dog class example, an implied metaclass can be used to add a function to the Dog class object to keep count of the dog population. The class definition file for the Dog class with an implied metaclass is shown in Figure 7-39.

```
#include "somobj.sc"

class:  Dog, local, class=DogClass_;

parent:  SOMObject;

data:
    char breed[100];
    int dogCount, class;

methods:
    void setBreed(char *myBreed);
    ---Set the Breed of this Dog.
    .
    .
    .
    override somInit, class;
    ---Initialize the dog counter to zero.

    SOMAny *somNew(), class;
    ---Increment the count of number of dogs.
    ---Create a dog object.

    int countDogs(), class;
    ---Return count of number of dogs.
```

*Figure 7-39. DOG.CSC—Implied Metaclass for the Dog Class*

In this example, the CLASS attribute is specified in both the Data and Method sections of the class definition. The CLASS attribute in the method prototypes indicates that the methods are new or overrides to existing class methods defined in the metaclass (SOMClass) of the parent of the Dog class. The CLASS attribute in the data declarations indicates that the data is associated with the class methods.

The Class section also contains a prefix specification (DogClass_) for class methods that are defined in this class definition file. The implementation of the Dog class and its implied metaclass, as shown in Figure 7-40, is similar to the previous examples .

```
#define Dog_Class_Source

#include "dog.ih"
#undef SOM_CurrentClass
#define SOM_CurrentClass \ DogClassData.parentMtab

#include <string.h>
#include <stdio.h>

/*  Same Dog Methods as before       */

#undef SOM_CurrentClass
#define SOM_CurrentClass \ M_DogClassData.parentMtab

SOM_SCOPE void SOMLINK DogClass_somInit(M_Dog *somSelf)
  {
   M_DogData *somThis = M_DogGetData(somSelf);
   parent_somInit(somSelf);
   _dogCount = 0;
  }

SOM_Scope SOMAny* SOMLINK DogClass_somNew(M_Dog *somSelf)
  {
   M_DogData *somThis = M_DogGetData(somSelf);
   -dogCount++;
   return parent_somNew(somSelf);
  }

SOM_Scope int SOMLINK DogClass_countDogs(M_Dog *somSelf)
  {
   M_DogData *somThis = M_DogGetData(somSelf);
   return _dogCount;
  }
```

*Figure 7-40. Implementation of an Implied Metaclass for the Dog Class*

SOM_CurrentClass is used to separate the implementation of the instance methods from the implementation of the class methods. The class methods defined in DOG.CSC are prefixed with DogClass_, as indicated in the Class section of the DOG.CSC file.

When an implied metaclass is used to define a subclass of a parent's metaclass, SOM defines a metaclass that is separate from either the class or its parent's metaclass. The class data, macros, methods, and functions related to this class have the prefix, M_. For example, somSelf is defined as a pointer to an instance of

the metaclass, M_Dog; that is, to the Dog class object. A client of the Dog class with an implied metaclass is shown in Figure 7-41 on page 7-32.

```
#include "dog.h"

#include <stdio.h>

int main()
{
  Dog *dog1;
  Dog *dog2;

  SOMClass *DogMeta;

  dog1 = DogNew();
  dog2 = Dognew();

  DogMeta = _somGetClass(dog1);

  printf("Count of dogs:  %d\n",
     _countDogs(DogMeta));

  _somFree(dog1);
  _somFree(dog2);
  return 0;
}
```

*Figure 7-41. A Client of the Dog Class with an Implied Metaclass*

The client program defines DogMeta as a pointer to a class object. The _somGetClass method inherited from the parent (SOMObject) of the Dog class is used to get a pointer to the Dog class object. Finally, the class method, countDogs is invoked by the Dog class object.

## Building SOM Class Libraries

Two basic rules explain the process of building SOM class libraries:

- .SC files for a class's metaclass, parent class, or ancestor class are required for the SOM compiler to process a class definition file. These files establish the relationships between classes.
- .H files are required for C-language compiler processing and building of the class library. These files resolve references to methods defined in other files.

With these rules in mind, the basic process for building a SOM class library can be summarized as follows:

1. Create .CSC files and do SOM compilation for all parent and ancestor classes and their related classes.
2. Create .CSC files for a class and its metaclass (if any).
3. Do a SOM compilation of the metaclass .CSC file.
4. Do a SOM compilation of the class .CSC file.
5. Do C-language compilations of .C implementation files, in any order.

## SOM ANIMALS Sample Program in the OS/2 2.0 Toolkit

The ANIMALS program introduces a programmer to basic SOM and object-oriented programming concepts. It defines six classes whose relationship corresponds to a zoological classification of animals. The classes and their relationships are shown in Figure 7-42.

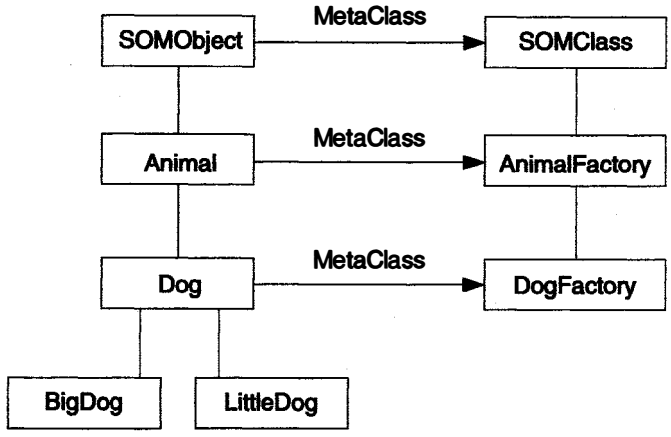



*Figure 7-42. Class Relationships in ANIMALS Sample Program*

The ANIMALS sample program is an expansion of the Dog class examples that have been used throughout this chapter. BigDog and LittleDog are subclasses of the Dog class. Dog is a subclass of the Animal class, which is a subclass of SOMObject. DogFactory is a subclass of AnimalFactory, which is a subclass of SOMClass. SOMClass and AnimalFactory provide constructor methods for the Animal class. SOMClass, AnimalFactory, and DogFactory provide constructor methods for the Dog, BigDog, and LittleDog classes.

The class definition of the dog classes in the ANIMALS sample program, includes the color of a dog, as well as the breed and sound it makes. Methods in the class definition file are grouped according to the type of function (for example, GetMethods, SetMethods, DisplayMethods, SystemMethodOverrides, AnimalMethodOverrides, parentOverrides).

## SOM ANIMALS Sample Program with Implied Metaclasses

The classes defined in the ANIMALS program can be restructured to use implied metaclasses. This eliminates the need for class definition files for the AnimalFactory and DogFactory metaclasses. The source code for the ANIMALS sample program with implied metaclasses is not included in the OS/2 2.0 Toolkit, but is provided in Figure 7-43 on page 7-34 through Figure 7-51 on page 7-44

```
#include <somobj.sc>

class: Animal, local, classprefix = m_;

parent: SOMObject;

passthru: C.h;
/* this is a passthru line */
endpassthru;

data:
  char *sound;
  int  nsound;
  int  cdata, classdata;

methods:

group: SetMethods;

   void setSound(char *mySound);
-- Tell the animal what kind of a sound it makes.

group: GetMethods;

   char *getGenus();
-- Returns the genus of animal.
-- This method should be overridden by derived classes.
-- The default version here just gives "unknown" as the genus.

   char *getSpecies();
-- Returns the species of animal.
-- This method should be overridden by derived classes.
-- The default version here just gives "unknown" as the species.

group: DisplayMethods;

   void talk();
-- Ask the animal to talk.  Normally this just prints the
-- string set by setSound(), but it can be overridden to do something
-- else.

   void display();
-- Displays an animal.  Derived classes should override this to
-- display new derived information, and then call their parent
-- version. Note: this method makes use of talk() to describe what
-- the animal says.
```

*Figure 7-43 (Part 1 of 2). ANIMAL.CSC—ANIMALS with Implied Metaclasses*

```
group: SystemMethodOverrides;

   override somInit, classmethod;
   override somInit;
-- This method will be invoked when a new animal is created.

   override somUninit;
-- This is invoked when an animal object is freed.

   override somDumpSelfInt;
-- Invoked when an animal is deleted.

group: Constructors;

   SOMAny * newAnimal(char *sound), classmethod;
-- Create an instance of an animal with a specific sound.

   int getCdata(),classmethod;
```

*Figure 7-43 (Part 2 of 2). ANIMAL.CSC—ANIMALS with Implied Metaclasses*

```
#define SOM_NoTest
#define Animal_Class_Source
#include "animal.ih"
#include <string.h>

SOM_Scope char *SOMLINK getGenus(Animal * somSelf)
{
    AnimalData *somThis = AnimalGetData(somSelf);
    AnimalMethodDebug("Animal", "getGenus");
    return ("Unknown Genus");
}

SOM_Scope char *SOMLINK getSpecies(Animal * somSelf)
{
    AnimalData *somThis = AnimalGetData(somSelf);
    AnimalMethodDebug("Animal", "getSpecies");
    return ("Unknown Species");
}
```

*Figure 7-44 (Part 1 of 3). ANIMAL.C—ANIMALS with Implied Metaclasses*

```
SOM_Scope void SOMLINK setSound(Animal * somSelf,
                                char *mySound)
{
    AnimalData *somThis = AnimalGetData(somSelf);
    AnimalMethodDebug("Animal", "setSound");

    if (_sound)
        (*SOMFree) (_sound);
    _sound = (char *) (*SOMMalloc) (strlen(mySound) + 1);
    strcpy(_sound, mySound);
}

SOM_Scope void SOMLINK talk(Animal * somSelf)
{
    AnimalData *somThis = AnimalGetData(somSelf);
    AnimalMethodDebug("Animal", "talk");
    somPrintf("%s\n", _sound);
}

SOM_Scope void SOMLINK display(Animal * somSelf)
{
    AnimalData *somThis = AnimalGetData(somSelf);
    AnimalMethodDebug("Animal", "display");
    somPrintf("  Genus: %s\n", _getGenus(somSelf));
    somPrintf("Species: %s\n", _getSpecies(somSelf));
    somPrintf("This Animal says\n");
    _talk(somSelf);
}

SOM_Scope void SOMLINK somInit(Animal * somSelf)
{
    AnimalData *somThis = AnimalGetData(somSelf);
    AnimalMethodDebug("Animal", "somInit");
    parent_somInit(somSelf);
    _sound = (char *) NULL;
}

SOM_Scope void SOMLINK somUninit(Animal * somSelf)
{
    AnimalData *somThis = AnimalGetData(somSelf);
    AnimalMethodDebug("Animal", "somUninit");
    if (_sound)
        (*SOMFree) (_sound);
    parent_somUninit(somSelf);
}
```

*Figure 7-44 (Part 2 of 3). ANIMAL.C—ANIMALS with Implied Metaclasses*

```
SOM_Scope void SOMLINK somDumpSelfInt(Animal * somSelf,
                                      int level)
{
    AnimalData *somThis = AnimalGetData(somSelf);
    AnimalMethodDebug("Animal", "somDumpSelfInt");
    _display(somSelf);
    parent_somDumpSelfInt(somSelf, level);
}

#undef SOM_CurrentClass
#define SOM_CurrentClass M_AnimalCClassData.parentMtab
SOM_Scope void   SOMLINK m_somInit(M_Animal *somSelf)
{
    M_AnimalData *somThis = M_AnimalGetData(somSelf);
    M_AnimalMethodDebug("M_Animal","m_somInit");

    _cdata = 1234;
    parent_somInit(somSelf);
}

/*
 *  Create an instance of an animal with a specific sound.
 */

SOM_Scope SOMAny *  SOMLINK m_newAnimal(M_Animal *somSelf,
                char *sound)
{
    M_AnimalData *somThis = M_AnimalGetData(somSelf);
    Animal *animal = _somNew(somSelf);
    M_AnimalMethodDebug("M_Animal","m_newAnimal");

    _setSound(animal, sound);
    return (animal);
}

#undef SOM_CurrentClass
#define SOM_CurrentClass M_AnimalCClassData.parentMtab
SOM_Scope int   SOMLINK m_getCdata(M_Animal *somSelf)
{
    M_AnimalData *somThis = M_AnimalGetData(somSelf);
    M_AnimalMethodDebug("M_Animal","m_getCdata");

    return (int) _cdata;
}
```

*Figure 7-44 (Part 3 of 3). ANIMAL.C—ANIMALS with Implied Metaclasses*

```
#include "animal.sc"
#include "somcls.sc"

class: Dog, local, classprefix = m_;

parent: Animal;

data:
  char *breed;
  char *color;
  char *classcolor,  classdata;

methods:

group: SetMethods;

   void setBreed(char *myBreed);
-- Set the breed of this dog.

   void setColor(char *myColor);
-- Set the color of this dog.

group: GetMethods;

   char *getBreed();
-- Get the breed of this dog.

   char *getColor();
-- Get the color of this dog.

group: AnimalOverrides;

   override getGenus;
-- This overrides the animal version.  This returns the genus
-- of a dog.

   override getSpecies;
-- This overrides the animal version.  This returns the species
-- of a dog.
```

*Figure 7-45 (Part 1 of 2). DOG.CSC—ANIMALS with Implied Metaclasses*

```
   override display;
-- Displays a dog.  This method displays the color and breed, and then
-- invokes the parent display to show information about the animal.
-- The parent display will invoke talk() to show what the animal says.
-- The version of talk() defined by Animal uses private Animal data
-- to determine how the animal talks.  But talk() is a method that will
-- be redefined by later derivations of Dog;  namely, LittleDog and
-- BigDog.  Both of these classes redefine the talk() method.  So
-- when a LittleDog invokes display(), which will be inherited from
-- this version, the following sequence occurs:
--   1. Dog::display() is invoked.
--   2. Dog::display() invokes Animal::display().
--   3. Animal::display() invokes talk(), which has been overridden
--      by LittleDog().
--   4. LittleDog::talk() uses a different algorithm for talking
--      than Animal::talk() uses.

group: SystemMethodOverrides;

   override somInit;
-- This method will be invoked when a new dog is created.
-- The reason this is automatically invoked is that Dog
-- is derived from Animal, and Animal uses SOMClassInit
-- as its metaclass.

   override somUninit;
-- This method is invoked when a Dog object is freed.

   override somDumpSelfInt;
-- Invoked when a dog is deleted.

   override somNew, classmethod;

group: Constructors;

   SOMAny * newDog(char *sound, char *breed, char *color), classmethod;
-- Create an instance of a dog with a specific sound, breed, and color.
```

*Figure 7-45 (Part 2 of 2). DOG.CSC—ANIMALS with Implied Metaclasses*

```
#define Dog_Class_Source

#include "dog.ih"
#include <string.h>
```

*Figure 7-46 (Part 1 of 4). DOG.C—ANIMALS with Implied Metaclasses*

```
SOM_Scope char *SOMLINK getBreed(Dog * somSelf)
{
    DogData *somThis = DogGetData(somSelf);
    DogMethodDebug("Dog", "getBreed");

    return _breed;
}

SOM_Scope char *SOMLINK getColor(Dog * somSelf)
{
    DogData *somThis = DogGetData(somSelf);
    DogMethodDebug("Dog", "getColor");

    return _color;
}

SOM_Scope void SOMLINK setBreed(Dog * somSelf, char *myBreed)
{
    DogData *somThis = DogGetData(somSelf);
    DogMethodDebug("Dog", "setBreed");

    if (_breed)
        (*SOMFree) (_breed);
    _breed = (char *) (*SOMMalloc) (strlen(myBreed) + 1);
    strcpy(_breed, myBreed);
}

SOM_Scope void SOMLINK setColor(Dog * somSelf, char *myColor)
{
    DogData *somThis = DogGetData(somSelf);
    DogMethodDebug("Dog", "setColor");

    if (_color)
        (*SOMFree) (_color);
    _color = (char *) (*SOMMalloc) (strlen(myColor) + 1);
    strcpy(_color, myColor);
}

SOM_Scope void SOMLINK somInit(Dog * somSelf)
{
    DogData *somThis = DogGetData(somSelf);
    DogMethodDebug("Dog", "somInit");
    parent_somInit(somSelf);

    _color = 0;
    _breed = 0;
}
```

*Figure 7-46 (Part 2 of 4). DOG.C—ANIMALS with Implied Metaclasses*

```
SOM_Scope void SOMLINK somUninit(Dog * somSelf)
{
    DogData *somThis = DogGetData(somSelf);
    DogMethodDebug("Dog", "somUninit");

    if (_color)
        (*SOMFree) (_color);
    if (_breed)
        (*SOMFree) (_breed);
    parent_somUninit(somSelf);
}
SOM_Scope void SOMLINK somDumpSelfInt(Dog * somSelf,
                                      int level)
{
    DogData *somThis = DogGetData(somSelf);
    DogMethodDebug("Dog", "somDumpSelfInt");
    parent_somDumpSelfInt(somSelf, level);
}

SOM_Scope char *SOMLINK getGenus(Dog * somSelf)
{
    DogData *somThis = DogGetData(somSelf);
    DogMethodDebug("Dog", "getGenus");
    return ("Canis");
}

SOM_Scope char *SOMLINK getSpecies(Dog * somSelf)
{
    DogData *somThis = DogGetData(somSelf);
    DogMethodDebug("Dog", "getSpecies");
    return ("Familiaris");
}

SOM_Scope void SOMLINK display(Dog * somSelf)
{
    DogData *somThis = DogGetData(somSelf);
    DogMethodDebug("Dog", "display");
    somPrintf("  Breed: %s\n", _getBreed(somSelf));
    somPrintf("  Color: %s\n", _getColor(somSelf));
    parent_display(somSelf);
}
```

*Figure 7-46 (Part 3 of 4). DOG.C—ANIMALS with Implied Metaclasses*

```
/*
 *  Create an instance of a dog with a specific sound, breed, and color.
 */

SOM_Scope SOMAny *  SOMLINK m_newDog(M_Dog *somSelf,
                char *sound,
                char *breed,
                char *color)
{
    M_DogData *somThis = M_DogGetData(somSelf);
    Dog *dog = _newAnimal(somSelf, sound);
    M_DogMethodDebug("M_Dog","m_newDog");

    _setBreed(dog, breed);
    _setColor(dog, color);
    return (dog);
}
```

*Figure 7-46 (Part 4 of 4). DOG.C—ANIMALS with Implied Metaclasses*

```
include "dog.sc"

class: BigDog, local;

parent: Dog;

methods:

group: parentOverrides;
   override talk;
-- Overrides Dog::talk().  See display() in Dog for a discussion
-- of how this changes the display() of a BigDog.
```

*Figure 7-47. BDOG.CSC—ANIMALS with Implied Metaclasses*

```
#define BigDog_Class_Source
#include "bdog.ih"

/*
 *  Overrides Dog::talk().  See display() in Dog for a discussion
 *  of how this changes the display() of a BigDog.
 */

SOM_Scope void   SOMLINK talk(BigDog *somSelf)
{
    /* BigDogData *somThis = BigDogGetData(somSelf); */
    BigDogMethodDebug("BigDog","talk");

    somPrintf("WOOF WOOF\n");
    somPrintf("WOOF WOOF\n");
    somPrintf("WOOF WOOF\n");
    somPrintf("WOOF WOOF\n");
    somSelf;
}
```

*Figure 7-48. BDOG.C— ANIMALS with Implied Metaclasses*

```
include "dog.sc"

class: LittleDog, local;

parent: Dog;

methods:

group: parentOverrides;

   override talk;
-- Overrides the Dog::talk().  See display() in Dog for a discussion
-- of how this changes the display() of a LittleDog.
```

*Figure 7-49. LDOG.CSC—ANIMALS with Implied Metaclasses*

```
#define LittleDog_Class_Source
#include "ldog.ih"

SOM_Scope void SOMLINK talk(LittleDog * somSelf)
{
/*  LittleDogData *somThis = LittleDogGetData(somSelf); */
    LittleDogMethodDebug("LittleDog", "talk");
    somPrintf("woof woof\n");
    somPrintf("woof woof\n");
    somSelf;   /* Avoid "Unreferenced parameter" compiler warning */
}
```

*Figure 7-50. LDOG.C—ANIMALS with Implied Metaclasses*

```
#include "animal.h"
#include "dog.h"
#include "ldog.h"
#include "bdog.h"

void main(int argc)
{
    Animal *myAnimal;
    Dog *myDog;
    LittleDog *myLittleDog;
    BigDog *myBigDog;

    /*
     * Create classes.
     */
    AnimalNewClass(Animal_MajorVersion, Animal_MinorVersion);
    DogNewClass(Dog_MajorVersion, Dog_MinorVersion);
    LittleDogNewClass(LittleDog_MajorVersion, LittleDog_MinorVersion);
    BigDogNewClass(BigDog_MajorVersion, BigDog_MinorVersion);

    SOM_TraceLevel = (argc > 1 ? 1 : 0);

    /*
     * Create objects using constructors
     */
    myAnimal = _newAnimal(_Animal, "Roar!!!");
    myDog = _newDog(_Dog, "Grrr", "Retriever", "Yellow");
    myLittleDog = _newDog(_LittleDog, "unknown noise",
                          "French Poodle", "Black");
    myBigDog = _newDog(_BigDog, "unknown noise",
                       "German Shepherd", "Brown");

    /*
     * Display objects.
     */
    somPrintf("====================\n");
    somPrintf("myAnimal:\n");
    _display(myAnimal);

    somPrintf("====================\n");
    somPrintf("myDog:\n");
    _display(myDog);

    somPrintf("====================\n");
    somPrintf("myLittleDog:\n");
    _display(myLittleDog);

    somPrintf("====================\n");
    somPrintf("myBigDog:\n");
    _display(myBigDog);
```

*Figure 7-51 (Part 1 of 2). MAIN.C—Client of ANIMALS with Implied Metaclasses*

```
    /*
     * Free objects.
     */
    somPrintf("\n");
    _somFree(myAnimal);
    _somFree(myDog);
    _somFree(myLittleDog);
    _somFree(myBigDog);

    somPrintf("Test successfully completed\n");
    exit(0);
}
```

*Figure 7-51 (Part 2 of 2). MAIN.C—Client of ANIMALS with Implied Metaclasses*

## Summary

*Table 7-10. Summary of SOM and Object-Oriented Programming Terminology*

| Term | Definition |
|---|---|
| Object | Set of data and actions that can be performed on that data. |
| Class | Generic description of sets of objects and their behavior. |
| Class Object | Run-time implementation of a class. |
| Instance Object | An object belonging to a certain class. |
| Method | Actions that can be performed on an object. |
| Instance Method | Actions that can be performed on instances of class objects. |
| Class Method | Actions that can be performed on class objects; also known as factory methods. |
| Metaclass | Class of a class. Defines class methods for a class object. |
| Implied Metaclass | Subclassing metaclass of parent class without separate CSC for resultant metaclass. |
| Client Program | Application that creates and manipulates instances of classes. |
| OIDL | Object Interface Definition Language; specification language for SOM class definitions. |
| Inheritance | Implications of derivation of new classes from existing classes. Child classes inherit methods of parent and ancestor classes. |
| Subclass | Child classes that define new behavior in terms of new methods not inherited from parent and ancestor classes. |
| Polymorphism | Different implementations of the same method for two or more classes. |
| Method Override | Polymorphism in SOM; child classes can override the implementation of methods inherited from parent and ancestor classes. |
| Encapsulation | The private, or internal view, of an object that describes how data and methods are actually implemented. |

# Chapter 8. Workplace Programming Interface

This chapter describes:

- CUA Guidelines for an Object-Oriented User Interface
- The OS/2 2.0 Object-Oriented User Interface: The Workplace
- The OS/2 2.0 Workplace Programming Interface

To establish a base definition of an object-oriented user interface, CUA guidelines are highlighted. The OS/2 Workplace Shell is an object-oriented user environment based on these guidelines. General object-oriented user interface concepts described in CUA guidelines are then used to explain the underlying design of the Workplace Programming Interface. For complete information on CUA guidelines, see *Systems Application Architecture: Common User Access Guide to User Interface Design* in the OS/2 2.0 Technical Library.

Because Workplace objects are built using the IBM System Object Model (SOM), the OS/2 2.0 Workplace Programming Interface requires a knowledge of SOM. This chapter assumes a knowledge of object-oriented programming and design concepts, as well as SOM. For more information on writing object-oriented programs using SOM, see Chapter 7, "Object-Oriented Programming Using SOM" on page 7-1. For reference information on the Object Definition Interface Language used to construct SOM classes, the predefined SOM classes and methods, and the SOM compiler, see the *System Object Model Guide and Reference* in the OS/2 2.0 Technical Library.

So that you can develop your own Workplace objects, class definition files for Workplace objects supported by the Shell are provided with the Toolkit. For detailed descriptions of Workplace classes and methods and the Workplace API, see the *Presentation Manager Programming Reference, Volume II* in the OS/2 2.0 Technical Library.

## CUA Guidelines for an Object-Oriented User Interface

Prior to 1991, CUA guidelines defined the user interface for *application-oriented* environments. In application-oriented environments, such as OS/2 1.X, the user performs tasks by starting an application. The user starts an application by selecting an installed program from a list of programs displayed in a window or by entering the program name at a command-line prompt. For example, the user can start an editor to create and edit a file, start a spreadsheet application to create and update a spreadsheet, or can send a file to a printer to print it.

In 1991, CUA guidelines defined the user interface for object-oriented environments. In object-oriented environments, such as OS/2 2.0, the user performs tasks by manipulating onscreen objects. The user does not start an application to perform a task. Instead, the user can:

- select an action, or task, that can be performed on an object. For example, the user selects the *open* action on an ASCII file object. The operating system then starts an editor to work on the file.
- select the *open* action on a spreadsheet file. The operating system then starts a spreadsheet application to work on the file.
- *drag* the ASCII file object to a printer object and *drop* it on the printer object. The file is then sent to the printer.

The concept of applications is transparent to the user. The user interacts with objects rather than with the operating system or with separate applications. The user focuses on the task and not on the tools used to perform the task. The user interacts with objects in the same manner across tasks.

## Objects, Classes, Hierarchies, and Inheritance

An object is any visual component of the user interface with which the user can work, independent of other items, to perform a task. An object can be represented by one or more graphic images, called icons. The user can interact with an object (or its icon) just as the user can interact with objects in the real world.

## Views of Objects

The user can also interact with an object by opening a window that displays more information about the object. The content of a window is a view of an object. A view is a way of looking at an object's information. Different views display information in different forms, just as information about an object is presented in the real world. An object can have more than one view.

The appearance of a window's contents and the kinds of interaction possible in a window are determined, in part by the type of view presented in the window. CUA guidelines describe four basic types of views:

- Composed Views

  A composed view of a data object arranges the object's data in an order that conveys the data's meaning. If the data is arranged differently in a composed view, the object has a different meaning. For example, a graph or chart object is typically displayed in a composed view because the arrangement of the components determines the meaning of the object as a whole. If the arrangement of the components changes, the meaning of the object changes.

- Contents Views

  A contents view lists the components of an object. The components can be ordered or unordered in the view; the order of the information displayed in a contents view does not affect the meaning of the object containing the information. CUA guidelines describe two kinds of contents views:

  - Icons view

    An icons view displays each object as an icon. Its purpose is to give the user an easy way to change the position of objects or to otherwise directly manipulate them.

    An object usually is represented by only one icon. However, for some tasks, the user might find it convenient to represent an object with more than one icon. For example, the user might want a representation of a printer object in more than one place so that the user can have easy access to the printer from anywhere. The user can create an additional icon, known as a shadow, to represent the same printer object.

  - Details view

    A details view combines small icons with text that provides additional information about objects. The type of information displayed depends on the type of object and the type of tasks the user wants to perform. A details view gives the user access to some of the object's more frequently used information, without requiring the user to open the object. Small icons are included in a details view to provide a way for the user to easily recognize objects and to directly manipulate each object.

- Settings Views

  A settings view displays information about the characteristics, attributes, or properties of an object, and provides a way for the user to change the settings of some characteristics or properties. A settings view is typically provided for each type of object.

- Help Views

  A help view provides information that can assist the user in working with an object. The type of information displayed in a help view depends on the type of help the user requests. For example, the user can request help for an entire window or for part of a window.

## Classes of Objects

Objects can be classified by similarities in characteristics and behavior. Each class of objects has a primary purpose that distinguishes it from other classes, and all three types of objects can contain other objects. CUA guidelines define three object classes:

- Container Objects

  A container object holds other objects. Its primary purpose is to provide a way for the user to group related objects for easy access and retrieval.

- Data Objects

  Data objects convey information, such as text or graphics, audio or video information.

- Device Objects

  A device object often represents a physical object in the real world. For example, a mouse object can represent the user's pointing device, and a modem object can represent the user's modem. Some device objects represent a logical object in the user's computer system rather than a physical object. For example, a shredder object can represent a logical object that disposes of the user's other objects. Some device objects can contain other objects. For example, a printer object can contain a queue of objects to be printed.

## Object Relationships

An object class can be defined in terms of another object class. It can be derived from another class, inheriting the same characteristics and behavior of the other class, yet having characteristics and behavior of its own. The class that it is derived from is called its parent class; the class itself is referred to as a subclass of its parent class.

The inheritance relationship between objects is hierarchical. An object that is lower in the inheritance hierarchy than another object has all of the characteristics of the object or objects above it and can have new characteristics of its own.

Other object relationships can also be hierarchical. For example, objects can be arranged in a containment hierarchy that illustrates which objects can contain which other objects.

## Interaction with Objects

In an object-oriented user environment, users interact with objects to perform tasks. The object-action paradigm is a pattern for interaction in which the user first selects an object, then selects an action. When the user selects an object, the system can then present a list of actions that can be applied to that object. Some actions may require the user to respond to additional choices. Interaction with an object through choices and controls is known as *indirect manipulation*.

At any given time an object has a set of actions that can be performed on it. Different objects have different actions that can be performed on them. Action choices for an object are displayed in pop-up menus that appear next to an object when the user presses the appropriate key or mouse button. The content of a pop-up menu is based on the object's context, which includes its current state, its location, and its contents.

Users may also interact with objects by way of a pointing device. This is known as *direct manipulation*. This interaction technique closely resembles the way the user interacts with objects in the real world. For example, the user can *pick up* an object and put it into a folder. This is also known as *dragging* an object from one place and *dropping* it at another place.

"Drag and drop" often involves a *source object* and a *target object*. A source object is usually the object the user is working with, and a target object is usually an object to which the user is transferring information; for example, if the user drags a file object to a printer object so that the file can be printed. The file is the source object and the printer is the target object.

The result of "drag and drop" can change depending on what the source object is and what the target object is. For example, if the user drags a file object from one folder object and drops it on another, the file is moved to the target folder. However, if the user drops the same file onto a printer, the operating system makes a copy of the file and puts the copy into the printer's queue to be printed. The original file is returned to its original location.

## Designing an Object-Oriented User Interface

The following items are key in designing an object-oriented user interface:

- Objects and their relationships
- Visual representations of objects
- Interaction Techniques and Mechanisms

Objects and their relationship can be defined by answering the following questions:

- What objects does the user require?
- How are the objects related?
- What properties and behaviors should the objects have?

To illustrate their importance, consider the example of the design of a software model of a car dealership. A salesperson needs a car object to represent each car model on his lot. He also needs a customer object to represent each customer that purchases a car. He needs a worksheet object to track sales, profits, inventory, customers, and so on. Finally, he needs container objects to group these objects.

The visual representations of objects must ensure consistency with one another and with the operating system. Visual representations of objects should address the functional aspects of visual representations, such as usability and purpose: Does the visual convey the purpose of the object being represented? Visual representations should also address the aesthetic aspects, such as shape, size, and color: Is the representation visually pleasing?

Users should interact with similar objects in similar ways and in ways that seem natural. Users should also have a choice of interaction mechanisms that suits their tasks, their level of skill, and their preferred style of interaction. In the car dealership example, a salesperson must be able to place information into a worksheet object in any of several ways. The salesperson can place a car object on top of the worksheet object, thereby transferring information about the car to the worksheet; or the salesperson can type information directly into the worksheet object; or the salesperson can select portions of information from the car object and copy them to the worksheet object.

## Defining the Objects for a Software Model

In order to design a software model that serves the needs of the users, the above is translated into:

- Who are the users?
- What tasks do they perform?

In the car dealership example, the salesperson is the user. The user's tasks can include:

- Determining what a customer wants, needs, and can afford
- Determining what cars are in stock that match the customer's wants, needs, and budget
- Negotiating an agreement using a worksheet
- Getting approval from the sales manager
- Giving the worksheet information to the finance manager

From this analysis, identify the objects that should be part of our software model can be identified. These include: a car object, a car lot object, a customer object, a customer list object, a worksheet object, a worksheet list object, a salesperson object, a sales manager object, and a finance manager object.

To simplify this discussion, consider the car object only. Each car object represents a real car for sale in the car lot. A car object contains descriptive information about the corresponding real car, such as its year, make, model, price, factory-installed options, color, and vehicle identification number. Because the primary purpose of a car object is to convey information, the car object is a data object.

## Determining Object Relationships and Behaviors

The next step in the design of a software model is to determine how each object interacts with other objects. In the car dealership, information contained in car objects must be transferable to a worksheet object. The user should have the option to drag a car object and drop it onto a worksheet object. The user should also have the option to choose an action to copy the information from a car object to a worksheet object.

### Determining the Necessary Views

After identifying and defining the objects, consider what views of the car object will give the user (salesperson) the best access to the objects and the information they contain. Car information consists of a combination of text (model, year, and so forth) and graphic information (a picture of a car) that make up a single, General Information View.

### Determining the Action Choices

From a salesperson's perspective, most of the information about a car object is fixed—that is, the information is based on a real-world object and cannot be changed unless something changes about the real-world object. For example, it would not make sense to allow a salesperson to change the color of a car object to correspond to the color of the actual car he is trying to sell. Because a salesperson can change little about a car object, the car object has only a few action choices:

- Open as general information
- Print
- Edit
- Copy to clipboard
- Find
- Windows

## The OS/2 Object-Oriented User Interface: The Workplace Shell

In OS/2 1.X, the Desktop is a collection of windows or icons representing windows associated with applications. In OS/2 2.0, the Desktop is a collection of objects (icons) and windows associated with those objects. The Desktop (which is also an object), the objects that appear on the Desktop, and the underlying code supporting these objects constitute the OS/2 Workplace Shell, the default user interface for OS/2 2.0.

The OS/2 Workplace Shell provides an object-oriented user environment that is based on the 1991 CUA guidelines. It provides a *seamless* environment, where all services are task-oriented and the user is shielded from the complexities of the operating system. The user can perform tasks faster and easier and with a shorter learning curve.

In the OS/2 Workplace, the Shell applications from OS/2 1.X are replaced by objects and collections of objects, or folders. Users do not have to be aware of where an object is (which drive or network) or what it is called. They can place an object wherever they wish and call it by any name. They do not have to know about EXEs, DLLs, device drivers, or how to add a printer or use a network. If they want to print a report on the laser printer down the hall, they can simply drag the icon representing the report and drop it onto the icon representing the laser printer which is labelled "Laser printer down the hall."

Users act on all objects in a consistent manner. They can act on program files in the same manner as program references. There is no difference in using programs on a network server or on a hard disk or on a CD drive. There is no difference in setting up or printing to a local or remote printer.

In the OS/2 Workplace, user's are not aware of the file system. They need not know anything about disk drives or directories. They need only know about folders and the objects they put into them. They can put applications, files, and so forth, in a folder. They can arrange things, regardless of where they physically reside, to suit themselves and their own needs. And they can label the folders by any name.

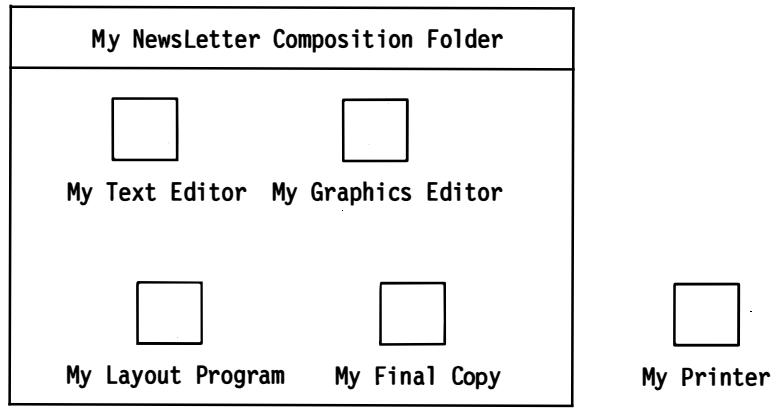



*Figure 8-1. Objects in a Folder*

Some of the objects that the OS/2 Workplace provides are described in Table 8-1. After installation of OS/2 2.0, some of them appear directly on the Desktop. Some of them are contained in folders. Users can rearrange and relabel them to suit themselves.

*Table 8-1. Some Workplace Objects Provided by OS/2 2.0*

| Object | Description |
|---|---|
| Clock | Set and view the current date, time, and alarm. |
| Country | Set and view international conventions for system elements (country, date, time, numbers). |
| Keyboard | Set and view keyboard configuration (timing, mappings, special needs). |
| Mouse | Set and view behavior of mouse device (timing, setup, button mappings). |
| Scheme Palette | Set and view window color and font attributes. |
| Font Palette | Set and view fonts for textual elements of user interface and apply fonts to windows. |
| Color Palette | Set and view colors for visual elements of user interface and apply color to a window. |
| Printer | Set and view a print destination (a print queue and its associated port. |
| Shredder | Destroy an object. |
| Sound | Enable/disable warning beep. |
| Special Needs | Explain implications of special needs mode when activated. |
| Spooler | Enable/disable spooling. Set and view spool path. |
| System | Set and view behavior of system elements (confirmations, logo, windows). |

## The OS/2 2.0 Workplace Programming Interface

While object-oriented user interfaces share some concepts with object-oriented programming, user objects may not necessarily correspond to software objects. Object-oriented programming can make the development of an object-oriented user interface easier. However, an object-oriented user interface can be developed with more traditional programming languages and tools.

The OS/2 2.0 Workplace is an example of a user interface developed using object-oriented programming, specifically, the IBM System Object Model. In fact, every user object in the OS/2 2.0 Workplace is an instance of a Workplace software class object. There is a one-to-one correspondence between Workplace (user) objects and Workplace (software) classes. This is evident in the class hierarchy for Workplace objects, as shown in Figure 8-2.

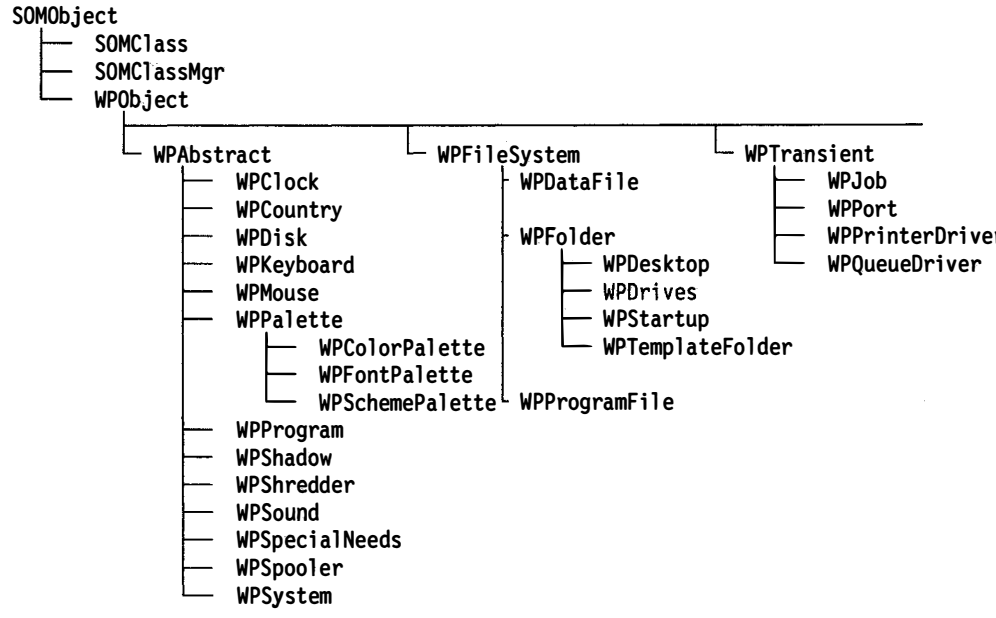



*Figure 8-2. Workplace Object Class Hierarchy*

All OS/2 2.0 Workplace classes are derived from a root Workplace class, WPObject, which is derived from the root SOM class, SOMObject. Workplace classes are defined using SOM's Object Interface Definition Language (OIDL). Workplace class libraries are built using the SOM compiler. Workplace objects are instantiated by the Workplace on behalf of the user through the Workplace Class List Object, installation programs, or batch files. The same rules that apply to SOM classes apply to Workplace classes, with the exception that applications cannot call Workplace methods. (SOM client applications can call SOM methods.)

Some Workplace classes (WPObject, WPAbstract, WPFileSystem, WPTransient) cannot be instantiated. These classes are provided as base classes which define common characteristics and behavior for descendant classes. Object characteristics and behavior are defined as methods for the object's class, as well as methods inherited from ancestor classes.

WPObject is the root class for all Workplace classes: it defines behavior common to all Workplace objects. The immediate descendants of WPObject are storage classes, from which all other Workplace classes are derived. *Storage classes* define methods for storing and retrieving data associated with instances of descendant classes. Storage classes provided with the OS/2 2.0 Workplace are shown in Table 8-2.

*Table 8-2. Workplace Storage Classes*

| Class | Storage of Object Instance Data |
|---|---|
| WPAbstract | Object instance data stored in user profile (OS2.INI). |
| WPFileSystem | Object instance data stored in files in the file system. |
| WPTransient | Object instance data not saved. |

Objects whose instance data and state is preserved between system shutdown and system startup are called *persistent objects.* Objects whose instance data and state need not be preserved between system shutdown and system startup are called non-persistent objects.

## Designing Workplace Classes

To design a Workplace class, first identify all the actions to which an object instance can respond. Based on this list, define the methods in the class definition file that correspond to the actions that were identified. To illustrate this process and understand how method requirements for a class are identified, consider the WPObject and WPAbstract classes.

Based on the general description of user objects in a CUA-conforming user environment, objects have:

- Properties (for example, an icon representation on the Workplace)
- Views that contain information about the object
- Context, or pop-up, menus that describe actions to which the object can respond
- Mobility (they can be directly manipulated)

These characteristics and behaviors should be reflected in the methods in the class definition file for the WPObject class. Table 8-3 shows the mapping of characteristics and behaviors common to all Workplace objects to method groups defined for this class.

*Table 8-3. Defining Methods for the WPObject Class*

| Method Group | Characteristic/Behavior |
|---|---|
| Settings-Notebook Methods | Properties |
| Save/Restore State Methods | Persistence of Object Instance Data |
| Object Usage Methods | Object Usage Information |
| Pop-up Menu Methods | Actions that users can perform on objects |
| Set/Query Object Information Methods | Object information (views, style, title) |
| Error Handling Methods | Error Returns |
| Memory Management Methods | Memory Allocation |
| Set/Cleanup Methods | Initialization and Termination |
| Direct Manipulation Methods | Mobility |

In Table 8-3, there are Method Groups in the WPObject class definition that support general object characteristics and behaviors that are included in the list above. There are also Method Groups in the WPObject class definition that support object characteristics and behaviors that are not included in the list.

## Settings-Notebook Methods

The WPObject class provides for a standard page in a Workplace object's Settings Notebook: a General page. This page describes all the general object properties (title, icon, and a user-definable setting to specify whether or not this object is a template). All Workplace objects have general properties associated with them; therefore, all Workplace objects have a General page in their Settings Notebook. Settings Notebook pages for Workplace objects are inherited from the ancestors of the Workplace class. This means that they include pages that have been added or removed by ancestor classes, in addition to the General page inherited from the root WPObject class.

For example, suppose that MyObject is a persistent object derived from the WPAbstract class. Because WPAbstract is derived from the WPObject class, MyObject inherits characteristics and behaviors from WPAbstract and WPObject. WPAbstract inherits its Settings Notebook from WPObject. MyObject, therefore, also inherits WPObject's Settings Notebook. WPObject defines a General page in the Settings Notebook for all classes of objects. Also, suppose that MyObject class defines two additional pages in its Settings Notebook: MyPage_1 and MyPage_2. That is, the Settings Notebook for the MyObject class has three pages: General, MyPage_1 and MyPage_2. Now suppose that YourObject class is derived from MyObject class and, therefore, by inheritance, defines a General page, as well as MyPage_1 and MyPage_2 in its Settings Notebook. Also suppose that YourObject class defines an additional page in its Settings Notebook: YourPage. The Settings Notebook for YourObject Class has four pages: General (inherited from the WPObject class), MyPage_1, MyPage_2 (inherited from MyObject class), and YourPage. The pages in the Settings Notebooks for MyObject and YourObject are shown in Figure 8-3.

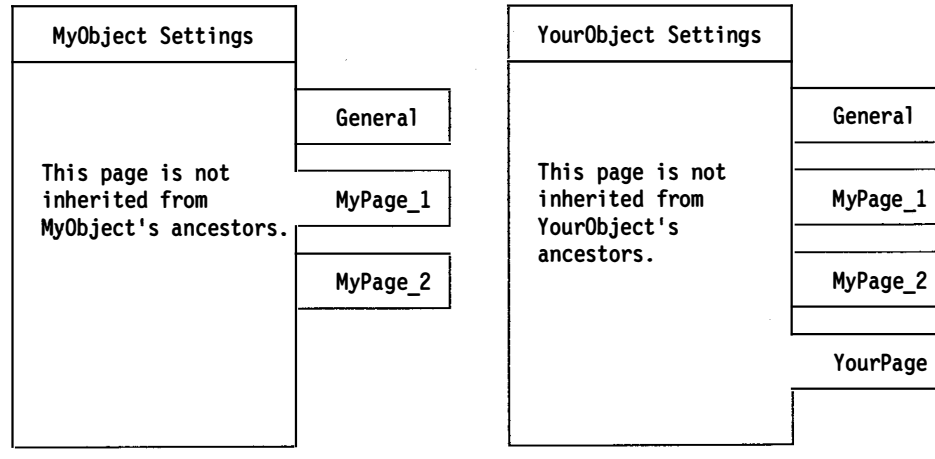



*Figure 8-3. Adding Pages to an Object's Settings Notebook*

The Settings-Notebook methods, as shown in Table 8-4 on page 8-11, allow you to add new pages or remove pages that a class has inherited from its ancestor's Settings Notebook.

| *Table 8-4. Settings Notebook Methods* | |
|---|---|
| **Method** | **Description** |
| wpAddObjectGeneralPage | Add the General page to the object's settings notebook. |
| wpAddSettingsPages | Adds pages to the object's settings notebook. |
| wpInsertSettingsPage | Insert a page into the object's settings notebook. |

An object's Settings Notebook is built by the Shell by calling the object's wpAddSettingsPages method. To add pages pages to the settings notebook that a class inherits from its ancestors, override this method.

Adding pages to the Settings Notebook a class has inherited from its ancestors is accomplished by overriding the wpAddSettingsPages method and calling a new method that inserts the new page. The new method calls the wpInsertSettingsPage method to insert the new page into the object's notebook. This technique is shown in Figure 8-4.

```
/*********************    New Methods     ***************************/
SOM_Scope ULONG SOMLINK MyObject_wpAddAnotherPage(MyObject *somSelf,
                HWND hwndNotebook)
{
PAGEINFO pageinfo;
.
/* Set up page information data structure for my new page.          */
.
/* Add the page to the Settings Notebook                            */
return _wpInsertSettingsPage(somSelf, hwndNotebook, &pageinfo);
}


/********************  Method Overrides  ***************************/

SOM_Scope BOOL32 SOMLINK MyObject_wpAddSettingsPage (MyObject *somSelf,
                HWND hwndNotebook)
{
.
.
if (parent_wpAddSettingsPage (somSelf, hwndNotebook)
   && _wpAddAnotherPage (somSelf, hwndNotebook))
   return (TRUE);        /*pages added successfully       */
else
   return (FALSE);       /*something failed -return error */
}
```

*Figure 8-4. Adding Pages to an Object's Settings Notebook*

New pages for an object's Settings Notebook can be placed at the top or at the bottom of pages inherited from the object's ancestor classes. By calling the parent_wpAddSettingsPage method before calling the new wpAddAnotherPage method, the new page is added to the top of the Settings Notebook, above pages inherited from ancestor classes. If the sequence is reversed, the new page is added to the bottom of the Settings Notebook, below pages inherited from ancestor classes.

A page can be removed from an object's Settings Notebook by overriding the ancestor's method that inserts it. From the previous example, the Settings Notebook for YourObject class inherits the pages (General, MyPage_1, and MyPage_2) defined by its ancestors (MyObject and WPObject). But YourObject class may have requirements for MyPage_1, but not MyPage_2. To remove MyPage_2 from YourObject's Settings Notebook, YourObject must override the method inherited from MyObject that adds MyPage_2 to the Settings Notebook and return SETTINGS_PAGE_REMOVED without calling the parent method. This is illustrated in Figure 8-5.

```
/********************* Method Overrides   *************************/
SOM_Scope ULONG SOMLINK YourObject_wpAddAnotherPage(YourObject *somSelf,
                HWND hwndNotebook)
{
.
.
/* Remove this page from the Settings notebook.                    */
return (SETTINGS_PAGE_REMOVED);
}
```

*Figure 8-5. Removing a Page from an Object's Settings Notebook*

The same technique can be used to replace or remove the General page from an object's Settings Notebook by overriding both the wpAddSettingsPages and wpAddObjectGeneralPage methods. The override to wpAddSettingsPages calls the wpAddObjectGeneralPage method. To remove the General page, the override to wpAddObjectGeneralPage returns SETTINGS_PAGE_REMOVED without calling the parent method. To replace the General page with another page, the override to wpAddObjectGeneralPage calls wpInsertSettingsPage without calling the parent method.

## Pop-Up Menus

Pop-up menu methods support the actions that the user can perform on an object. These actions appear in a context, or pop-up menu, when the user presses button 2 of the pointing device. A pop-up menu contains action choices for an object in its current context, or state. The contents of a pop-up menu depends on the state of the object.

Pop-up menus consist of a set of selectable items and any pull-down, or conditional cascade, menus associated with them. In Figure 8-6, **Open**, **Help** and **Print** are items in the object's primary pop-up menu. **Settings** and **Details** are items in the **Open** pull-down or conditional cascade menu. Conditional cascade menus have mini-push buttons displayed next to the pop-up menu item. If the user selects the pop-up menu item, a default action listed in the pull-down menu is performed. If the user selects the mini-pushbutton, the pull-down menu is displayed. It is a conditional cascade menu because it is displayed only if the user selects the mini-push button.

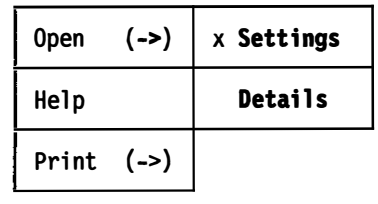



*Figure 8-6. Pop-Up and Conditional Cascade Menus*

Like Settings Notebook pages, pop-up menus are inherited from a class's ancestor classes. This means that they include pop-up menu items that ancestor classes have added to or removed from the pop-up menu inherited from WPObject. The Pop-up Methods, as shown in Table 8-5, allow you to add new menu items to or remove menu items from the pop-up menu inherited from an object's ancestor classes.

*Table 8-5. Pop-Up Menu Methods to Modify an Object's Pop-up Menu*

| Method | Description |
|---|---|
| wpFilterPopupMenu | Filter out options from object's pop-up menu that don't apply. |
| wpInsertPopupMenuItems | Insert items into object's pop-up menu. |
| wpModifyPopupMenu | Add new options to the object's pop-up menu. |

When a user requests an object's pop-up menu, the Shell builds the object's pop-up menu by calling the object's wpFilterPopupMenu and wpModifyPopupMenu methods. The wpInsertPopupMenuItems method is called by an override to the wpModifyPopupMenu method to add new options to an object's pop-up menu.

***Adding and Removing Items from a Pop-Up Menu:*** The WPObject class defines a set of standard pop-up menu items that are inherited by all Workplace objects. The pop-up menu of a Workplace object consists of a subset of the standard pop-up menu items and any new menu items defined for the object's class or inherited from other ancestors.

Workplace classes can add or delete standard pop-up menu items from their pop-up menu by overriding the wpFilterPopupMenu method. Each standard pop-up menu item is associated with a flag defined by the WPObject class, as shown in Table 8-6.

*Table 8-6. Flags for Standard Pop-Up Menu Items Inherited from WPObject*

| Menu Item Flag | Description |
|---|---|
| CTXT_CREATEANOTHER | Create another |
| CTXT_OPEN | Open |
| CTXT_CLOSE | Close |
| CTXT_SETTINGS | Open settings |
| CTXT_PRINT | Print |
| CTXT_HELP | Help |
| CTXT_DELETE | Delete |
| CTXT_COPY | Copy |
| CTXT_MOVE | Move |
| CTXT_SHADOW | Create Shadow |
| CTXT_WINDOW | Window |

The wpFilterPopupMenu method returns the flags that represent the pop-up menu items for the object. In removing a standard pop-up menu item from the pop-up menu, the override to wpFilterPopupMenu masks the flag that corresponds to the item being removed from the flags that represent the standard pop-up menu items inherited from the object's parent. For example, suppose that printing MyObject has no meaning. To remove the **Print** option from MyObject's pop-up menu, wpFilterPopupMenu is overridden as shown in Figure 8-7.

```
/*                    Method Override                              */

/*   Filter out any options from the pop-up that don't apply. */

SOM_Scope ULONG   SOMLINK MyObject_wpFilterPopupMenu(MyObject *somSelf,
                          ULONG ulFlags, HWND hwndCnr, BOOL32 fMultiSelect)
{
MyObjectData *somThis = MyObjectGetData(somSelf);
MyObjectMethodDebug("MyObject","MyObject_wpFilterPopupMenu");

/* Don't allow anyone to print MyObject                           */

return( parent_wpFilterPopupMenu(somSelf,ulFlags,hwndCnr,fMultiSelect)
                              & ~CTXT_PRINT );
}
```

*Figure 8-7. Removing Standard Items from an Object's Pop-Up Menu*

In Figure 8-7, flags that represent the standard pop-up menu items of MyObject's parent class are returned from the call to parent_wpFilterPopupMenu. To remove the **Print** option from MyObject's pop-up menu, these flags are AND'd with the complement of CTXT_PRINT. Conversely, if the pop-up menu of MyObject's parent class did not include the **Print** option, the **Print** option can be added to MyObject's pop-up menu by OR'ing these flags with CTXT_PRINT.

**Usage Note**

An object's pop-up menu is inherited from its ancestors. To ensure that calls to the wpFilterPopupMenu method belonging to the object's ancestors do not add the menu item after it is deleted or remove the menu item after it is added, the parent_wpFilterPopupMenu method is called first.

Other Workplace classes provided with the Shell define standard pop-up menu items for their descendants. Figure 8-8 lists the flags associated with these additional standard pop-up menu items.

| WPFolder | WPDesktop | WPProgram | WPPalette |
|---|---|---|---|
| CTXT_ICON | CTXT_SHUTDOWN | CTXT_PROGRAM | CTXT_PALETTE |
| CTXT_TREE | CTXT_LOCKUP | | |
| CTXT_DETAILS | | | |
| CTXT_FIND | | | |
| CTXT_SELECT | | | |
| CTXT_ARRANGE | | | |
| CTXT_SORT | | | |

*Figure 8-8. Flags for Standard Pop-Up Menu Items Defined by Other Workplace Classes*

These items are added by the respective classes using the wpModifyPopupMenu method. Any override to wpModifyPopupMenu must call the parent method so that these items are added to the pop-up menu inherited from ancestor classes.

***Adding Class-Specific Items to the Primary Pop-Up Menu:*** New items are added to the pop-up menu inherited from an object's ancestors by overriding the object's wpModifyPopupMenu method and calling the wpInsertPopupMenuItems method. The object's class also defines a new FilterPopupMenu method that returns flags representing the new items added to the object's pop-up menu.

For example, to add **New Item** to MyObject's pop-up menu, the new menu item is defined in a resource file in the same manner as menus are defined in PM programs. An ID is assigned to the new menu and to the menu item, as shown in Figure 8-9.

```
#define ID_MOREITEMS   WPMENUID_USER+1
#define ID_NEWITEMS    WPMENUID_USER+2


MENU ID_MOREITEMS LOADONCALL MOVEABLE DISCARDABLE
BEGIN
  MENUITEM "New Item", ID_NEWITEMS
END
```

*Figure 8-9. Resource File Defining New Items for Object's Pop-Up Menu*

IDs for class-specific menus and menu items have a value greater than WPMENUID_USER so they do not conflict with IDs for menus and menu items defined by the Workplace classes provided with the Shell. IDs for standard items in an object's pop-up menu are shown in Figure 8-10.

| **WPObject** | **WPFolder** |
|---|---|
| WPMENUID_OPEN | WPMENUID_SELECT |
| WPMENUID_HELP | WPMENUID_SORT |
| WPMENUID_PRINT | |

*Figure 8-10. IDs for Standard Items in a Pop-Up Menu*

A new method for MyObject class, wpMyFilterPopupMenu, returns flags (in this case, a flag) that represent the new items added to the inherited pop-up menu for the MyObject class. The override to wpModifyPopupMenu uses the flags returned from wpMyFilterPopupMenu to determine which items are inserted in MyObject's pop-up menu. This is illustrated in Figure 8-11 on page 8-16.

```
/* Define MYCTXT_NEWITEM here                                        */

/*                        New Method                                 */
/*   Filter new items added to MyObject's pop-up menu                */
SOM_Scope ULONG  SOMLINK MyObject_wpMyFilterPopupMenu(MyObject *somSelf,
                      ULONG ulFlags, HWND hwndCnr, BOOL32 fMultiSelect)
{
.
.
return(MYCTXT_NEWITEM);
}

/*                      Method Override                              */
/*    Add a new item to MyObject's pop-up menu                       */

SOM_Scope BOOL32   SOMLINK MyObject_wpModifyPopupMenu(MyObject *somSelf,
                         HWND hwndMenu, HWND hwndCnr, ULONG ulPosition)
{
.
/* Get flags for new items for MyObject's pop-up menu               */
flags=_wpMyFilterPopupMenu(...);

/* Insert new items in MyObject's Primary Menu if flag is set  */
if (flags & MYCTXT_NEWITEM)
   _wpInsertPopupMenuItemsA(somSelf, hwndMenu, ulPosition, hmod,
                            ID_MOREITEMS, WPMENUID_PRIMARY);

/* Add the items inherited from MyObject's parent      */
return (parent_wpModifyPopupMenu(somSelf,hwndMenu,hwndCnr,ulPosition));
}
```

*Figure 8-11. Adding Class-Specific Items to an Object's Pop-Up Menu*

The wpInsertPopupMenuItems method requires a handle to the module where the menu resource is defined, the ID for the menu resource, and the ID for the menu where the item is being inserted. In the above example, ID_MOREITEMS is the ID for the menu resource that defines the new menu item being added to the object's primary pop-up menu. WPMENUID_PRIMARY is the ID for the object's primary pop-up menu, where **New Items** is being inserted.

An item can be added to a pop-up menu submenu, or conditional cascaded menu by specifying the ID for the conditional cascaded menu on the call to the wpInsertPopupMenuItems method. For example, to add **New Items** to the **Open** conditional cascaded menu, the call to wpInsertPopupMenuItems is modified as shown in Figure 8-12.

```
/* Insert new items in MyObject's Open SubMenu if flag is set  */
if (flags & MYCTXT_NEWITEM)
   _wpInsertPopupMenuItemsA(somSelf, hwndMenu, ulPosition, hmod,
                            ID_MOREITEMS, WPMENUID_OPEN);
```

*Figure 8-12. Adding an Item to a Pop-Up Menu Submenu*

***Removing Class-Specific Items from an Object's Pop-Up Menu:*** Class-specific pop-up menu items inherited from ancestor classes are removed by overriding the filtering methods that return flags that represent those items. For example, suppose MyObject is the parent class of YourObject, but YourObject has no requirement for the **New Item** in the pop-up menu it inherits from MyObject. To remove **New Item** from YourObject's pop-up menu, YourObject's class overrides wpMyFilterPopupMenu method and does not return the MYCTXT_NEWITEM flag MyObject defined for **New Item**. This technique, as shown in Figure 8-13 is similar to that described for removing standard items from an object's context menu. It requires that MyObject publish wpMyFilterPopupMenu method, as well as the MYCTXT_NEWITEM flag so that subclasses of MyObject can remove or add **New Item.**

```
/*                    Method Override                              */
/*   Filter new items added to MyObject's pop-up menu              */
SOM_Scope ULONG  SOMLINK YourObject_wpMyFilterPopupMenu(
  YourObject *somSelf, ULONG ulFlags, HWND hwndCnr, BOOL32 fMultiSelect)
{

return(0);    /* Return no flags so that 'New Item' is not        */
              /* included in YourObject's pop-up menu             */
}
```

*Figure 8-13. Removing Class-Specific Items from a Pop-Up Menu*

***Adding Conditional Cascaded Menus to the Primary Pop-Up Menu:*** Items on an object's pop-up menu sometimes have pulldown menus or submenus associated with them. In the Workplace, pop-up submenus are conditional cascaded menus, as shown in Figure 8-14.

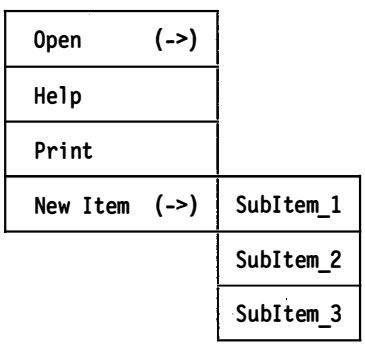



*Figure 8-14. Conditional Cascaded Menus*

In the previous example, **New Item** is not a pulldown menu. However, **New Item** can be defined as a pulldown menu by defining it as a submenu in MyObject's resource file, as shown in Figure 8-15 on page 8-18.

```
#define ID_MOREITEMS    WPMENUID_USER+1
#define ID_NEWITEMS     WPMENUID_USER+2
#define ID_SUBITEM1     WPMENUID_USER+3
#define ID_SUBITEM2     WPMENUID_USER+4
#define ID_SUBITEM3     WPMENUID_USER+5

MENU ID_MOREITEMS LOADONCALL MOVEABLE DISCARDABLE
BEGIN
  SUBMENU "New Item", ID_NEWITEMS
    BEGIN
      MENUITEM "SubItem_1"      ID_SUBITEM1
      MENUITEM "SubItem_2"      ID_SUBITEM2
      MENUITEM "SubItem_3"      ID_SUBITEM3
    END
END
```

*Figure 8-15. Resource File Defining Pulldown for Object's Pop-Up Menu*

The **New Item** submenu is added to MyObject's primary pop-up menu using the same technique as shown in Figure 8-11 on page 8-16. For the Shell to display the submenu as a conditional cascade menu with the mini-push button and default selection, the menu's style and default selection must be set, as shown in Figure 8-16.

```
/*                    Method Override                          */
/*     Add a new item to MyObject's pop-up menu                */

SOM_SCOPE BOOL32 SOMLINK MyObject_wpModifyPopupMenu(MyObject *somSelf,
                         HWND hwndMenu, HWND hwndCnr, ULONG iPosition)
{
.
MENUITEM mi;
.
.
.
/* Get a handle to the New Item submenu                        */
WinSendMsg(hwndMenu, MM_QUERYITEM,
           MPFROM2SHORT(ID_NEWITEMS), (MPARAM)&mi);
hwndSubMenu=mi.hwndSubMenu;

/* Query the menu's style                                      */
ulstyle=WinQueryWindowULong(hwndSubMenu, QWL_STYLE);

/* Add conditional cascade capabilities to existing menu style */
ulstyle |= MS_CONDITIONALCASCADE;

/* Set menu style to include conditional cascade capabilities  */
WinSetWindowULong(hwndSubMenu, QWL_STYLE, ulstyle);

/* Set the default selection in the submenu - it must exist    */
WinSendMsg(hwndSubMenu, MM_SETDEFAULTITEMID, (MPARAM)ID_SUBITEM1, 0L);
.
.
}
```

*Figure 8-16. Creating a Conditional Cascaded Menu for a Pop-Up Menu Item*

***Supporting User Selection of New Pop-Up Menu Items:*** When a class defines new actions for its pop-up menu, it must provide for the processing of the actions when the user selects the action. This is done by overriding the Pop-Up Menu Methods shown in Table 8-7.

*Table 8-7. Pop-Up Menu Methods that Support New Pop-Up Menu Items*

| Method | Description |
|---|---|
| wpMenuItemHelpSelected | Display the help associated with class-specific pop-up menu item. |
| wpMenuItemSelected | Process class-specific pop-up menu item. |

Using the previous example, MyObject supports SubItem_1 by overriding the wpMenuItemSelected method as shown in Figure 8-17.

```
/*                    Method Override                          */
/* Process input from the extra menu option that we added. */

SOM_Scope void   SOMLINK MyObject_wpMenuItemSelected(MyObject *self,
                         HWND hwndFrame, ULONG MenuID)
{
  .
  /* Which of our menu items was selected ?  */
  switch( MenuId )
  {
     case ID_SUBITEM1:
          .
     case ID_SUBITEM2:
          .
     case ID_SUBITEM3:
          .
     case ID_SUBITEM4:
          .
     default:
          parent_wpMenuItemSelected
  }
```

*Figure 8-17. Supporting User Selection of New Pop-Up Menu Items*

MyObject can support help for new pop-up menu items by overriding the wpMenuItemHelpSelected method in a similar manner.

***Support for User Selection of Standard Pop-Up Menu Items:*** When the user selects a standard action from a pop-up menu, the Shell calls one of the Pop-Up Menu methods shown in Table 8-8.

*Table 8-8. Pop-Up Menu Methods Supporting Standard Pop-Up Menu Items*

| Method | Description |
|---|---|
| wpClose | Close all open views of an object. |
| wpCopyObject | Create a new copy of the object. |
| wpCreateFromTemplate | Create an object from a template. |
| wpCreateShadowObject | Create a shadow of an object. |
| wpDelete | Delete an object and prompt for confirmation if necessary. |
| wpDisplayHelp | Display a help panel. |
| wpHide | Hide or minimize open views of an object. |
| wpRestore | Restore hidden or minimized views of an object. |
| wpMoveObject | Move the object to a different location. |
| wpOpen | Open a view of the object. |
| wpPrintObject | Print a view of the object. |

***Open Views:*** Objects can have Open Actions or Open Views associated with them. Open Views are typically the views of an object (for example, icon, details, and settings). Open Views (for data file objects) also include programs or program references that the user has associated with the object. Open Views are displayed when the user selects the cascade mini-push button that appears next to the **Open** action on the pop-up menu. The user can then select the default Open View or any of the Open Views that are listed in the conditional cascade menu.

The Workplace defines a set of predefined Open Views for Workplace objects, as shown in Table 8-9. Some of the predefined Open Views are meaningful only to certain Workplace classes: OPEN_RUNNING is meaningful only to a program or program reference object; OPEN_TREE is meaningful only to file system objects such as folders, drives and directories.

*Table 8-9. Predefined Open Views for Workplace Objects*

| View | Description |
|---|---|
| OPEN_CONTENTS | Open content view. |
| OPEN_DEFAULT | Open default view. |
| OPEN_DETAILS | Open details view. |
| OPEN_HELP | Open help view. |
| OPEN_RUNNING | Execute object. |
| OPEN_SETTINGS | Open settings notebook. |
| OPEN_TREE | Open tree view. |
| OPEN_USER | Open class-specific view (value greater than OPEN_USER) |

Workplace classes can define new Open Views for their objects by:

1. Adding the **New View** menu item to the Open submenu.
2. Overriding the wpMenuItemSelected method to support user selection of the **New View** menu item.
3. Overriding the wpOpen method to open the **New View.**
4. Creating and opening a standard window for the **New View** by calling WinCreateStdWindow.

   **Note:** The preferred method for displaying application views of an object is for the object to start a separate process (using DosExecPgm) for the application. This approach moves the larger part of the application code out of the Shell's process, thus conserving the Shell's resources. It also helps prevent a misbehaved application from potentially interfering with the execution of the Shell.
5. Adding a USAGE_VIEW item to the object's in-use list by calling the wpAddToObjUseList method.
6. Registering the **New View** by calling the wpRegisterView method.

The CAR Sample Workplace Object in the Toolkit demonstrates how to define and open a new Open View.

***Object Shadows:*** A shadow of an object is an object that refers to another object's data. A shadow object can often appear to be a copy of the object that it refers to. However, it is not a copy but another representation of the object. This means that when data in a shadow object is changed, the same data is changed in the original object and all references to it. This is in contrast to a copy of an object, such as a data file, where the data in the copy can be changed and not affect the original.

Some objects cannot be copied and only a shadow can be created in order to have that object reside in multiple folders. These typically are device objects. For example, because there is only one physical mouse device in the system, there cannot be two different mouse objects.

A shadow is created by the user by direct manipulation (holding CTRL and SHIFT keys down while dragging/dropping the object) or by selecting the **Create Shadow** on the pop-up menu. In both cases, the Shell responds by calling the object's wpCreateShadowObject method.

***Helps for Objects:*** Help for Workplace objects is provided in the same manner as for PM applications using the Information Presentation Facility (IPF). Help instances are created for and associated with objects by the Shell. Help panels for windows and window controls are created using the IPF tag language and compiled by the IPF compiler into a Help Library. The Help Library is placed in the systems's \OS2\HELP directory or in another directory in the LIBPATH. Help panels are associated with windows and window controls by help tables defined in the object's resource file. See the *Information Presentation Facility Guide and Reference* in the OS/2 2.0 Technical Library for for more information on creating Help panels and Help libraries.

Help for Workplace objects is supported by the object's wpDisplayHelp, wpQueryDefaultHelp and wpMenuItemHelpSelected methods. The wpQueryDefaultHelp and wpMenuItemHelpSelected methods are overridden to support class-specific views and menu items.

## Object Information Methods

Object Information Methods allow you to set and query information (default help and view, details, icon and icon, style, and title) associated with an object. The methods in this group set and query object information as shown in Table 8-10.

*Table 8-10. Object Information Methods*

| Methods | Operate on |
|---|---|
| wpSet/wpQueryDefaultHelp | Default help panel for object |
| wpSet/wpQueryDefaultView | Default view for object |
| wpQueryDetailsData | Current details data for object |
| wpSet/wpQueryIcon | Current icon for an object |
| wpSet/wpQueryIconData | Current icon and icon data for an object |
| wpSet/wpQueryStyle | Current style of an object |
| wpSet/wpQueryTitle | Current title of an object |

***Object Styles:*** Just as PM windows have class styles, Workplace objects have object (instance) styles, as shown in Table 8-11. Object instance styles define object behavior. They can be changed after an object has been created by calling the wpSetStyle method.

*Table 8-11. Object Class Styles*

| Style | Description |
|---|---|
| OBJSTYLE_NOCOPY | Cannot be copied. |
| OBJSTYLE_NODELETE | Cannot be deleted. |
| OBJSTYLE_NODRAG | Cannot be dragged. |
| OBJSTYLE_NOSHADOW | Cannot have shadow created. |
| OBJSTYLE_NOMOVE | Cannot move. |
| OBJSTYLE_NOPRINT | Cannot be printed. |
| OBJSTYLE_NOTDEFAULTICON | Destroy icon when object goes dormant. |
| OBJSTYLE_TEMPLATE | Object is a template. |
| OBJSTYLE_NOTVISIBLE | Object is hidden. |
| OBJSTYLE_NORENAME | Object cannot be renamed. |

***Object Templates:*** An object template is the primary user mechanism for creating new instances of objects. Specifically, a template is a state of an object where the default drag operation is "Create another," that is, dragging and dropping the template results in the creation of an instance of the object. The visual representation of a template is the object's icon on top of a "yellow sticky pad" with the top sheet slightly peeled up. Any object that supports the "Create another" action can be changed by the user to and from a template state by selecting the template check box on the General page in the object's settings notebook. A template is created automatically when a class is registered, unless the class wpclsQueryStyle class method returns CLSSTYLE_NEVERTEMPLATE.

When the operating system is first installed, template objects reside in the Templates folder on the Desktop. The templates folder always contains a template object for each class of object installed on the system that supports the "Create Another" action. Any new object registered by the WinRegisterObjectClass function that supports the "Create Another" action automatically appears in this folder. A template for each object class registered using this function cannot be removed from the templates folder.

***Object Details:*** Some Workplace classes define a set of information that a user can display in a details view of all instances of objects belonging to the class. A details view is a Presentation Manager container control window. For more information on Presentation Manager container control windows, see *OS/2 Programming Guide, Volume II* in the OS/2 2.0 Technical Library.

A details view, as represented in Figure 8-18, consists of a window with data arranged in columns which have headings. A row of data, also known as a record, provides information for a specific instance of an object belonging to the class. Each element in the details view can be text, icons, or bit maps. OWNERDRAW of elements is also supported.

Details for Instances of MyObject Class

| | Item1 | Item2 | Item3 | Item4 | Item5 | Item6 | Item7 | Item8 |
|---|---|---|---|---|---|---|---|---|
| Instance 1 | | | | | | | | |
| Instance 2 | | | | | | | | |
| | | | | | | | | |

*Figure 8-18. Details View for a Workplace Class*

The column headings for a details view are specified by overriding the wpclsQueryDetailsInfo class method. A record that contains information for an object instance of the class is constructed by overriding the wpQueryDetailsData instance method.

An object can inherit the set of details defined by its ancestors. A record containing details information for an object, therefore, can contain sets of details for the object that are defined by its parent, its grandparent, and so forth. To meet this requirement, the Workplace constructs records as contiguous blocks of memory.

A record is built for an object by the override to the wpQueryDetailsData instance method. To inherit the details defined by ancestor classes, wpQueryDetailsData calls the parent class's wpQueryDetailsData method, which calls its parent class's wpQueryDetailsData method, and so forth through the oldest ancestor that defines details data. With each call to wpQueryDetailsData, a variable length block of memory containing a set of details for the object is added to the record. A pointer is also moved to the end of the last block of memory added to the record where the next call to wpQueryDetailsData adds the next block of memory. As shown in Figure 8-19 on page 8-24, each block in the record contains a set of details for the object defined by its class and/or by one of its ancestor classes.

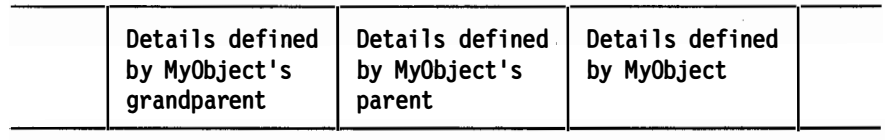



*Figure 8-19. A Details Record for an Object*

The format for details information contained in a record is defined in an override to the wpclsQueryDetailsInfo class method. The format is defined in a linked list of CLASSFIELDINFO data structures for each details data item in a record. This linked list is constructed in the same manner as a record: wpclsQueryDetailsInfo calls its parent class's wpclsQueryDetailsInfo method, which calls its parent class's wpclsQueryDetailsInfo method, and so forth. As shown in Figure 8-20, each call adds a set of CLASSFIELDINFO data structures to the linked list until the linked list contains CLASSFIELDINFO data structures for each details data item in the object's details record.

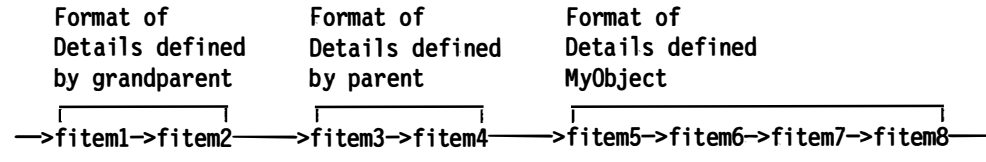



*Figure 8-20. Format for Items in a Details Record for an Object*

The CLASSFIELDINFO data structure is defined in the header files provided with the OS/2 2.0 Toolkit. It describes the attributes of details data in a particular column. It includes the title for the column heading and the format of the data. The flags used to specify the attribute of the title for the field and the data within the field are the same as those used in the container control window's FIELDINFO data structure.

The CAR Sample Workplace Object in the OS/2 2.0 Toolkit can be used to illustrate the requirements for defining details data for a class of objects. CAR can provide details data for CAR objects by:

1. Defining a data structure for the items to be included in the details view. as shown in Figure 8-21.

```
/* CARDETAILS:  Structure used for details view  */

typedef struct _CARDETAILS {
  PSZ      pszMake;     // Manufacturer
  PSZ      pszModel;    // Model
  PSZ      pszColor;    // Color of car
  CDATE    cdateSale;   // Date of sale
  ULONG    ulPrice;     // Price in dollars
} CARDETAILS;
typedef CARDETAILS *PCARDETAILS;
```

*Figure 8-21. Defining a Structure for CAR Details*

2. Defining and initializing a static array of CLASSFIELDINFO structures for each detail, as shown in Figure 8-22. This is done on class initialization, when wpclsInitData is called.

   **Note:** "Title" and "Icon" details are defined for CAR objects by its ancestor class WPObject. This means that CLASSFIELDINFO structures for "Title" and "Icon" details are defined and initialized in the WPObject class definition.

```
#define NUM_CAR_FIELDS    5

CLASSFIELDINFO fieldinfo[NUM_CAR_FIELDS];

PSZ pszCarColTitles[] = {"Make", "Model","Color","Sale date","Price ($)"};

.
.
//
// METHOD: wpclsInitData                                    ( ) PRIVATE
//                                                          (X) PUBLIC
// PURPOSE:
//   Initialize the details data
//
SOM_Scope void SOMLINK carM_wpclsInitData(M_Car *somSelf)
{
   USHORT          i;
   PCLASSFIELDINFO pCFI;

   /*  Call the parent class method first                          */
   parent_wpclsInitData(somSelf);

   /* Initialize everything needed for the CLASSFIELDINFO structures  */
   /* for the CAR object class                                        */
   for (i=0,pCFI=fieldinfo;i<NUM_CAR_FIELDS;i++,pCFI++)
   {
      /* memset the structure to zero's                               */
      memset((PCH)pCFI,0,sizeof(CLASSFIELDINFO));

      pCFI->cb        = sizeof(CLASSFIELDINFO);
      pCFI->flData    = CFA_RIGHT | CFA_SEPARATOR | CFA_FIREADONLY;
      pCFI->flTitle   = CFA_CENTER | CFA_SEPARATOR | CFA_HORZSEPARATOR |
                        CFA_STRING | CFA_FITITLEREADONLY;
      pCFI->pNextFieldInfo = pCFI + 1;
      pCFI->pTitleData = pszCarColTitles[i];
      pCFI->flCompare  = COMPARE_SUPPORTED;

      switch (i)
      {
         case 0:
            pCFI->flData        |= CFA_STRING;
            pCFI->offFieldData   = (ULONG)(FIELDOFFSET(CARDETAILS,pszMake));
            pCFI->ulLenFieldData = sizeof(PSZ);
            break;
         case 1:
            pCFI->flData        |= CFA_STRING;
            pCFI->offFieldData   = (ULONG)(FIELDOFFSET(CARDETAILS,pszModel));
            pCFI->ulLenFieldData = sizeof(PSZ);
            break;
```

*Figure 8-22 (Part 1 of 2). Initializing CLASSFIELDINFO Structures for CAR Details*

```
            case 2:
               pCFI->flData        |= CFA_STRING;
               pCFI->offFieldData   = (ULONG)(FIELDOFFSET(CARDETAILS,pszColor));
               pCFI->ulLenFieldData = sizeof(PSZ);
               break;
            case 3:
               pCFI->flData        |= CFA_DATE;
               pCFI->offFieldData   = (ULONG)(FIELDOFFSET(CARDETAILS,cdateSale));
               pCFI->ulLenFieldData = sizeof(CDATE);
               break;
            case 4:
               pCFI->flData        |= CFA_ULONG;
               pCFI->offFieldData   = (ULONG)(FIELDOFFSET(CARDETAILS,ulPrice));
               pCFI->ulLenFieldData = sizeof(ULONG);
               break;
         }
      }
      fieldinfo[NUM_CAR_FIELDS-1].pNextFieldInfo = NULL;
   }
```

*Figure 8-22 (Part 2 of 2). Initializing CLASSFIELDINFO Structures for CAR Details*

3. Defining the column headings and format of the details data placed into the object's details record by overriding the wpclsQueryDetailsInfo class method, as shown in Figure 8-23 on page 8-27.

```
//
// OVERRIDE: wpclsQueryDetailsInfo
//
// Let parent know the size of the RECORDCORE devoted to data from CAR
// if pSize is NON-NULL
//
// Link in the Car CLASSFIELDINFO chain if pp is NON_NULL
//
SOM_Scope ULONG SOMLINK redcarM_wpclsQueryDetailsInfo(M_Car *self,
                                                      PCLASSFIELDINFO *pp,
                                                      PULONG pSize)
{
/* M_CarData       *somThis = M_CarGetData(self); */
   ULONG           n_prev_cols;
   PCLASSFIELDINFO pCFI;
   ULONG           i;
   M_CarMethodDebug("M_Car","carM_wpclsQueryDetailsInfo");

   /* Always call the parent method first
    */
   n_prev_cols = parent_wpclsQueryDetailsInfo(self,pp,pSize);

   /* If the request was for the size of the car details column data,
    * size needed by the car details
    */
   if (pSize)
      *pSize += sizeof(CARDETAILS);

   /* If the request was for the chained fieldinfo structures,
    * link them in
    *
    * eventually the chain will look like
    *
    * Grandparent - Parent - Me - Child - Grandchild
    *
    * I will be getting the pointer to the beginning of the chain
    *
    * If the beginning of the chain is 0, I will assign the address
    * of my first CLASSFIELDINFO structure to *pp.
    * Otherwise *pp points to the first column description in the
    * chain.  We need to walk the chain and link our CLASSFIELDINFO
    * structures at the end.
    */
   if (pp)
   {
      /* Link our CLASSFIELDINFO structures to the chain
       * find the last link in the chain
       */
      if (pCFI = *pp)
      {
         for (i=0;i<n_prev_cols;i++)
             pCFI = (pCFI->pNextFieldInfo) ? pCFI->pNextFieldInfo : pCFI;

         pCFI->pNextFieldInfo = fieldinfo;
      }
      else
         *pp = fieldinfo;
   }
   return ((ULONG) (n_prev_cols + NUM_CAR_FIELDS));
}
```

*Figure 8-23. Defining Format of Details Data*

4. Adding details data to object's Details record by overriding the wpQueryDetailsData instance method, as shown in Figure 8-24.

```
//
// Append details data to the end of the RECORDCORE structure
//
SOM_Scope ULONG SOMLINK car_wpQueryDetailsData
                              (Car *self, PVOID *ppDetailsData)
{
   PCARDETAILS pCD;

   CarData *somThis = CarGetData(self);
   CarMethodDebug("Car","car_wpQueryDetailsData");

   parent_wpQueryDetailsData(self,ppDetailsData);

   pCD                 = (PCARDETAILS) *ppDetailsData;
   pCD->pszMake        = "Volkswagen";      /* Manufacturer     */
   pCD->pszModel       = "Cabriolet";       /* Model name       */
   pCD->pszColor       = "RED";             /* Color of the car */
   pCD->cdateSale.day  = 24;                /* Date of sale     */
   pCD->cdateSale.month = 12;
   pCD->cdateSale.year  = 91;
   pCD->ulPrice        = 14000;             /* Price in dollars */

   ((PBYTE) (*ppDetailsData)) += sizeof(CARDETAILS);

   return(TRUE);
}
```

*Figure 8-24. Appending Details Data to end of Object's Details Record*

## Direct Manipulation Methods

Direct Manipulation methods support dragging and dropping one object on another object. The object being dragged is the source object, and the object on which the source object is dropped is the target object. The Workplace Shell tracks a source object that the user drags, and notifies target objects and windows when the source object is being dragged over them and when it is dropped on them. For target objects in the Workplace, the Shell calls the target object's Direct Manipulation methods to process the source object being dragged and dropped.

For PM applications, the Shell sends DM_ messages to PM windows using the standard Drag and Drop protocol. The Shell will drag source objects rendered as OBJECT or as OS2FILE and will accept source objects rendered in the same way. The Shell also will send a DM_PRINTOBJECT message to items dropped on the printer object.

**Usage Note**

Users can drag source objects over windows that an object creates. When this occurs, the Shell sends DM_ messages to these windows. Therefore, window procedures associated with the windows that the object creates must be able to process the DM_ messages. For more information on direct manipulation in PM windows, see the *OS/2 Programming Guide, Volume II* in the OS/2 2.0 Technical Library.

Target objects are not necessarily able to process every type of source object that is dropped on them. They are, however, capable of processing more than one type of dropped source object. Printer objects, for example, cannot print binary files, but they can print both text files and graphics files. Because of differing capabilities, each target object should determine if it can process the source object being dropped on it.

*Table 8-12. DM Notifications from the Shell to Target Objects and to Windows*

| Notification | Objects | Windows |
|---|---|---|
| Format drag information (source) | wpFormatDragItem | None |
| Request rendering format (source) | wpRender | DM_RENDER |
| Rendering request completed (source) | wpRenderComplete | DM_RENDERCOMPLETE |
| Objects being dragged over (target) | wpDragOver | DM_DRAGOVER |
| Object has been dropped (target) | wpDrop | DM_DROP |
| Drag/Drop is complete (target) | wpEndConversation | DM_ENDCONVERSATION |

## Save/Restore State Methods

The Save/Restore State methods, as shown in Table 8-13 are important to the class definitions of persistent objects.

*Table 8-13. Save/Restore the Object's State Methods*

| Method | Description |
|---|---|
| wpSaveImmediate | Save the object's state. |
| wpSaveDeferred | Asynchronously saves the object's state. |
| wpSave/wpRestoreState | Save/Restore the object's state. |
| wpSave/wpRestoreData | Save/Restore blocks of instance data. |
| wpSave/wpRestoreLong | Save/Restore a 32-bit instance data value. |
| wpSave/wpRestoreString | Save/Restore an ASCII instance data string. |

When an object is awakened, wpRestoreState is called by the Shell. The wpSaveImmediate method calls wpSaveState. The Shell calls wpSaveImmediate and wpSaveState when an object is closed or made dormant or the system is shut down. The wpSaveImmediate method can also be called by an object's methods when a critical instance variable is changed. The wpSaveDeferred method asynchronously saves data and, for performance reasons, should be used in preference to the wpSaveImmediate method.

To save or restore data relevant to an object, override the wpSaveState and wpRestoreState methods. The override for wpSaveState calls wpSaveData, wpSaveLong, and wpSaveString, depending on the type of instance data associated with your object. Similarly, the override for wpRestoreState calls wpRestoreData, wpRestoreLong, and wpRestoreString.

## Object Usage Methods

Object Usage methods, as shown in Table 8-14, allow an object to keep track of its resources and how it is being used.

*Table 8-14. Object Usage Methods*

| Method | Description |
|---|---|
| wpAddToObjUseList | Add an item to the object's in-use list. |
| wpDeleteFromObjUseList | Remove an item from the object's in-use list. |
| wpFindUseItem | Retrieve an item from the object's in-use list. |

Every Workplace object in the system has an in-use list. The in-use list provides the object with information, such as the number of container windows into which the container has been inserted. It also provides the number of open views (contents and settings) of itself that already exist and how much memory it has allocated.

The in-use list is a linked list of USEITEM data structures. The USEITEM data structure consists of an item type and a pointer to the next USEITEM data structure, and is immediately followed by an item type-specific data structure. The types of items that can be added to an object's in-use list and the type-specific data structures that follow each USEITEM data structure are shown in Table 8-15.

*Table 8-15. Types of In-Use Items for Objects*

| Item | Description | USEITEM + |
|---|---|---|
| USAGE_MEMORY | Memory has been allocated by wpAllocMem method. | MEMORYITEM |
| USAGE_OPENVIEW | A view of the object has been opened. | VIEWITEM |
| USAGE_RECORD | The object has been inserted into a container window. | RECORDITEM |

The USEITEM, MEMORYITEM, VIEWITEM, and RECORDITEM data structures are shown in Figure 8-25 on page 8-31 and are defined in the header files that are provided with the Toolkit.

```
typedef struct _USEITEM {
   ULONG   type;                /* type of this item     */
   struct _USEITEM FAR *pNext; /* next item in use list */
}USEITEM;

typedef struct _MEMORYITEM{
   ULONG   cbBuffer;            /* Number of bytes in memory block */
}MEMORYITEM;

typedef struct _VIEWITEM{
   ULONG   view;                /* Object view that this represents */
   LHANDLE handle;              /* Open handle                      */
   ULONG ulViewState;           /* View state flags                 */
   HWND hwndCnr;                /* Used only by system              */
   PMINIRECORDCORE              /* Used only by system
}VIEWITEM;

typedef struct _RECORDITEM{
   HWND   hwndCnr;/* Container into which object is being inserted */
   PMINIRECORDCORE pRecord;     /* Record pointer within container  */
   ULONG  ulUser;               /* For application use              */
}RECORDITEM;
```

*Figure 8-25. Data Structures for Object In-Use Items*

The wpAddToObjUseList method adds items to the object's in-use list when the events outlined in Table 8-15 on page 8-30 occur. When memory is allocated for an object by its wpAllocMem method, wpAllocMem calls wpAddToObjUseList to add a USAGE_MEMORY item to the object's in-use list. When a view of an object is opened by the object's wpOpen method, this method is called to add a USAGE_OPENVIEW item to the object's in-use list. When an object is inserted into a container window, a USAGE_RECORD item is added to the object's in-use list by the object's wpCnrInsertObject method.

Conversely, when memory is freed by the object's wpFreeMem method, wpDeleteFromObjUseList is called by wpFreeMem to remove a USAGE_MEMORY item from the object's in-use list. When views are closed by the object's wpClose method, wpDeleteFromObjUseList removes a USAGE_VIEW item from the object's in-use list. When objects are removed from a container window by the object's wpCnrRemoveObject method, wpDeleteFromObjUseList removes a USAGE_RECORD item from the object's in-use list.

wpFindUseItem is used to determine how an object is currently being used. It searches an object's in-use list for items of a specified type and returns a pointer to the USEITEM structure that matches the specified type

## Setup/Cleanup Methods

Setup/Cleanup Methods, as shown in Table 8-16, support object initialization and destruction.

*Table 8-16. Setup/Cleanup Methods*

| Method | Description |
|---|---|
| wpFree | Destroy the object and deallocate its associated resources. |
| wpInitData | Allocate and Initialize the object's instance data. |
| wpScanSetupString | Parse the setup string that is specified when the object is created. |
| wpSetup | Change object characteristics and behaviors as specified in setup string. |
| wpUnInitData | Deallocate object's instance data. |

Classes can define KEYNAMES and values that affect the behavior of their objects. KEYNAME values can be specified in a setup string when an object is created by calling WinCreateObject or when a change in the behavior of an existing object is required and initiated by calling WinSetObjectData. Because KEYNAMES have default values, setup strings are not required for these calls.

Every class defines its own set of KEYNAMES and values. KEYNAMES and the values supported by the WPObject class are listed in Table 8-17.

*Table 8-17 (Page 1 of 3). WPObject KEYNAMES and Values*

| KEYNAME | Value | Description |
|---|---|---|
| TITLE | Title | Sets the object's title. Equivalent to calling the wpSetTitle method. |
| ICONFILE | filename | Sets the object's icon. Equivalent to calling the wpSetIconData method. |
| HELPPANEL | id | Sets the object's default help panel. Equivalent to calling the wpSetDefaultHelp method. |
| TEMPLATE | YES | User can create object template. Equivalent to calling wpSetStyle method with OBJSTYLE_TEMPLATE style. |
| | NO | User cannot create object template. |
| NODELETE | YES | User cannot delete object. Equivalent to calling wpSetStyle method with OBJSTYLE_NODELETE style. |
| | NO | User can delete object. |

*Table 8-17 (Page 2 of 3). WPObject KEYNAMES and Values*

| KEYNAME | Value | Description |
|---|---|---|
| NOCOPY | YES | User cannot copy object. Equivalent to calling wpSetStyle method with OBJSTYLE_NOCOPY style. |
| | NO | User can copy object. |
| NOMOVE | YES | User cannot move object. Equivalent to calling wpSetStyle method with OBJSTYLE_NOMOVE style. |
| | NO | User can move object. |
| NOSHADOW | YES | User cannot create shadow of object. Equivalent to calling wpSetStyle method with OBJSTYLE_NOSHADOW style. |
| | NO | User can create shadow of object. |
| NOTVISIBLE | YES | Object is not visible. Equivalent to calling wpSetStyle method with OBJSTYLE_NOTVISIBLE style. |
| | NO | Object is visible. |
| NOPRINT | YES | User cannot print object. Equivalent to calling wpSetStyle method with OBJSTYLE_NOPRINT style. |
| | NO | User can print object. |
| ICONRESOURCE | id,module | Sets the object's icon. Equivalent to calling wpSetIconData method. The "id" is the icon resource ID in the dynamic link library (DLL) "module." |
| ICONPOS | x,y | Sets the object's initial icon position in a folder. The x and y values represent the position in the folder in percentage coordinates. |
| OBJECTID | <name> | Sets a persistent ID for the object. The OBJECTID can be used to obtain a pointer or handle to the object by calling the wpclsQueryObject method or WinQueryObject function. An OBJECTID is any unique string preceded with a '<' and terminated with a '>.' |

*Table 8-17 (Page 3 of 3). WPObject KEYNAMES and Values*

| KEYNAME | Value | Description |
|---|---|---|
| NORENAME | YES | User cannot rename object. Equivalent to calling the wpSetStyle method with OBJSTAYLE_NORENAME style. |
| | NO | User can rename object. |
| NODRAG | YES | User cannot drag object. Equivalent to calling wpSetStyle method with OBJSTYLE_NODRAG style. |
| | NO | User can drag object. |
| VIEWBUTTON | HIDE | Views of object have a hide button instead of a minimize button. |
| | MINIMIZE | Views of object have a minimize button instead of hide button. |
| MINWIN | HIDE | Views of object are hidden when minimize button is selected. |
| | VIEWER | Views of object minimized to minimized window viewer when minimize button selected. |
| | DESKTOP | Views of object minimized to Desktop when minimize button selected. |
| CONCURRENTVIEW | YES | New views of object created every time user selects open. |
| | NO | Open views of object resurface when user selects open. |
| OPEN | SETTINGS | Open settings view when object is created or when WinSetObjectData is called. |
| | DEFAULT | Open default view when object is created or when WinSetObjectData is called. |

KEYNAMES and their values are specified in a setup string as shown in Figure 8-26.

```
pszSetupString="TITLE=MYObject;ICONFILE=MYOBJ.ICO;OBJECTID=<MyObjectID>";
```

*Figure 8-26. Example of Object Setup String*

KEYNAMES and their values are separated by semicolons in the setup string. The escape character (^) followed by a comma or semicolon can be used to represent a comma or semicolon if they are required in a KEYNAME value specification. KEYNAMES are processed by an object's wpSetup method, which is called when WinCreateObject and WinSetObjectData are called by an application. Classes that define their own KEYNAMES override the wpSetup method, as shown in Figure 8-27 on page 8-35. The override for wpSetup scans the setup string for its KEYNAMES and processes them.

```
wpSetup

    if (_wpScanSetupString("ICONFILE"...)
       _wpSetIconData();
    if (_wpScanSetupString("MYKEYNAME"..)
       do whatever;
```

*Figure 8-27. Processing KEYNAMES for a Class*

Because wpSetup is called as a result of WinCreateObject and WinSetObjectData, applications can effect changes to objects that already exist on the Desktop and are being used. For example, to effect changes to the icons for objects that already exist, an application calls:

1. WinQueryObject to get a handle to the object using the object's OBJECTID.
2. WinSetObjectData with a ICONDATA KEYNAME value specified in the setup string.

***The WPAbstract Class: Persistent Objects:*** By definition, WPAbstract and WPFileSystem objects are persistent objects. Because the the class examples used in this discussion are derived from the WPAbstract class, consider the characteristics of WPAbstract objects.

WPAbstract objects are identified by a numeric handle. Persistent objects need to keep track of where instances are, their attributes, and instance data. These requirements are reflected in the method overrides defined for this class. There are no new methods for the WPAbstract class. However, the WPAbstract class overrides some of the methods inherited from WPObject; methods that act on instance attributes, data and object state information. These include:

- wpCopyObject
- wpQueryTitle
- wpQueryIcon
- wpSetup
- wpSaveImmediate
- wpSaveState
- wpRestoreState
- wpMoveObject
- wpQueryIconData
- wpSetIconData

## Workplace Class Methods: Implied Metaclasses

All Workplace objects have implied metaclasses. This means that the object's metaclass is defined in the same file as the object's class. This means that a separate class definition file is not required for Workplace metaclasses. This is the primary advantage of implied metaclasses. The number of files to build an object is reduced.

Metaclasses define all the class methods for a class. Class methods act on class data common to all object instances of the class. Metaclasses are, therefore, the mechanisms for defining class properties, as opposed to instance properties. Workplace class properties include default attributes for all instances of the class, for example, the default object title, the default help panel, the default icon, and so forth.

Workplace class methods are prefixed by "wpcls." Some of the class methods defined by the WPObject class are shown in Table 8-18.

*Table 8-18. Some WPObject Class Methods*

| Method | Description |
|---|---|
| wpclsFindObjectEnd | End a search for an object belonging to the class. |
| wpclsFindObjectFirst | Begin a search for an object belonging to the class. |
| wpclsFindObjectNext | Find another object belonging to the class. |
| wpclsQueryDefaultHelp | Get the default help panel for instances of the class. |
| wpclsQueryDefaultView | Get the default open view for instances of the class. |
| wpclsQueryDetails | Get the default details view items for instances of the class. |
| wpclsQueryDetailsInfo | Get and set details information for instances of the class. |
| wpclsQueryIcon | Get the default icon for instances of the class. |
| wpclsQueryIconData | Get and set the default icon data for instances of the class. |
| wpclsQueryObject | Get pointer or handle to persistent object. |
| wpclsQueryStyle | Get the default object style for instances of the class. |
| wpclsQueryTitle | Get the default title for instances of the class. |

Default class characteristics are inherited by instances of the class unless the class overrides the methods that operate on those characteristics. For example, to define a default object style for instances of MyObject, MyObject overrides wpclsQueryStyle and returns the appropriate default class style. Default class styles for objects belonging to a class are shown in Table 8-19.

*Table 8-19. Default Class Styles for Objects*

| Style | Description |
|---|---|
| CLSSTYLE_NEVERMOVE | User cannot move object. |
| CLSSTYLE_NEVERSHADOW | User cannot create shadow for object. |
| CLSSTYLE_NEVERCOPY | User cannot copy object. |
| CLSSTYLE_NEVERTEMPLATE | User cannot create template for object. |
| CLSSTYLE_NEVERDELETE | User cannot delete object. |
| CLSSTYLE_NEVERPRINT | User cannot print object. |
| CLSSTYLE_NEVERDRAG | User cannot drag object. |
| CLSSTYLE_NEVERVISIBLE | Object instances are always invisible. |
| CLSSTYLE_NEVERRENAME | User cannot rename object. |

The FindObject class methods (wpclsFindObjectFirst, wpclsFindObjectNext, and wpclsFindObjectEnd) are similar in function to the DosFind functions in the OS/2 file system. The wpclsFindObjectFirst method initiates the search for objects belonging to the class with the specified characteristics. The wpclsFindObjectEnd method ends the search. They are typically overridden when a more robust FindObject function is required or when the user initiates a search for an object.

## Creating a Workplace Object: The Car Object

Many of the things previously discussed are demonstrated in the CAR Sample Workplace Object that is included in the Toolkit. CAR is a spin-off implementation of the car object example used in the *Systems Application Architecture: Common User Access Guide to User Interface Design* and highlighted in the first part of this chapter.

The CAR object has two views:

- An "Open car" view which is a representation of a car that moves randomly around a window.
- A "Settings" view or Settings notebook which allows the user to change the sound of the horn (on the "Horn Beep" page) and the speed of the car (on the "Dashboard" page).

The "Horn Beep" and "Dashboard" pages in CAR's settings notebook are dialog windows whose contents are defined in a dialog template in the CAR.RC resource file. The resources are appended to the binary CAR.DLL file. The help panels associated with the dialogs are defined in the CAR.HLP help library. The association between the help panels and the dialogs is established with the dialog template in the resource file.

CAR is a persistent object, but not a part of the file system. It is, therefore, derived from the WPAbstract class, which is derived from the WPObject root Workplace class. This means that CAR inherits the methods from WPAbstract, which in turn inherits all, and overrides some, of the instance and class methods from WPObject.

CAR defines new methods and overrides some instance and class methods inherited from both WPAbstract and WPObject. New methods defined for the CAR class are summarized in Table 8-20.

*Table 8-20. New Methods for CAR Class*

| Method | Description |
|---|---|
| carSetInfo | Sets up the car information. It is called by the dialog procedure to update the car data, as the user interacts with the dialog window. |
| carQueryInfo | Gets the car information. It is called by the dialog procedure to get the latest car data when the dialog window opened. |
| wpAddDashboardPage | Inserts the "Dashboard" page in the Settings Notebook. It is called by the wpAddSettingsPage method. |
| wpAddHornBeepPage | Inserts the "Horn Beep" page in the Settings Notebook. It is called by the wpAddSettingsPage method. |

The implementation of the CAR object consists primarily of overrides to instance and class methods inherited from the WPObject and WPAbstract classes:

- **Providing for Persistence of the CAR Object**

  Because the CAR class defines its own set of data, it overrides the wpSaveState and wpRestoreState methods. These method overrides, in turn, call the wpSaveLong and wpRestoreLong methods to save instance data defined for the CAR class.

- **Changing the Settings Notebook Pages**

  CAR creates Settings Notebook pages by overriding the wpAddSettingsPage method inherited from WPObject. The wpAddSettingsPage method calls the new methods defined by the CAR class, wpAddDashboardPage and wpAddHornBeepPage. In turn, each of these call the wpInsertSettingsPage method inherited from WPObject.

- **Changing the Context Menu**

  Because the user action "Print a car" does not make sense for this object, the CAR class removes the **Print** option from the CAR object's context menu by overriding the wpFilterPopupMenu inherited from WPOject.

- **Adding an Action to the Context Menu**

  The CAR class adds a "Beep horn" action to its context menu by overriding the wpModifyPopupMenu method inherited from WPObject. The method override for wpModifyPopupMenu calls the wpInsertPopupMenuItem, also inherited from WPObject, to insert this item into CAR's context menu.

- **Adding an Item to the Open Menu**

  The CAR class adds a **Open car** view (OPEN_CAR) to its "Open" menu by overriding its wpModifyPopupMenu method. The method override calls wpInsertPopupMenuItem to insert **Open car** in the "Open menu."

- **Creating and Registering a new Open View**

  A window for the OPEN_CAR view is created and opened by calling WinCreateStdWindow. The window procedure for the client of the OPEN_CAR window registers the OPEN_CAR view and associates the OPEN_CAR view with that window by calling the wpRegisterView method.

- **Processing New Menu Items on a Context or Pulldown Menu**

  When a class defines new actions for its menus, it must provide for the processing of the actions when the user selects the action. This is done by overriding its wpMenuItemSelected method inherited from WPObject. Depending on the action the user selects, the method override for wpMenuItemSelected calls the object's wpOpen method to open the OPEN_CAR view or calls the carBeepHorn method to beep the car's horn.

- **Processing Help for New Actions on Context Menu**

  When a class defines new actions for its context menu, it must also provide for the processing of the HELP for those actions when the user requests it. This is accomplished by overriding the wpMenuItemHelpSelected method inherited from WPObject. Given the action selected by the user, the method override for wpMenuItemHelpSelected calls the wpDisplayHelp method inherited from WPObject to display the help for that action. wpDisplayHelp requires the ID for the help panel associated with the action, as well as the name of the help library where it resides.

The implementation of the CAR object also demonstrates:

1. The use of a Release Order in CAR.CSC.
2. The use of external stem and prefix attributes in the Class section of CAR.CSC. Debugging is easier because method names are externalized.
3. The use of the message queue for the Shell. Object code runs on the Shell's thread. Windows associated with objects receive messages through the Shell's message queue. Objects do not need their own message queues.

## The Workplace Application Interface

Outside the Workplace environment, objects are DLLs which consist of data and code that operates on that data when objects are instantiated in the Workplace (runtime) environment. Workplace objects have no "life" outside the Workplace environment.

Workplace classes "come to life" when the class is registered with the Workplace Shell and the class is instantiated. The Workplace Shell and SOM provide the underlying code (predefined Workplace Object methods) that supports an object's existence. The Shell calls the appropriate object methods when the user interacts with the object. In this sense, the Shell is the client of all Workplace objects. The Shell manipulates the object (its code) on behalf of its users.

Applications, on the other hand, cannot call Workplace object methods directly. They are not clients of Workplace objects, in the same sense that applications can be clients of SOM objects. Workplace objects are derived from the WPObject class, which is derived from the SOMObject class. They share all the features of SOM objects: inheritance, polymorphism, and so forth. But only the Shell can directly manipulate them.

Because there are times when applications may need to effect changes to the Desktop and to objects on the Desktop, the Workplace Shell provides functions that allow you to effect those changes. These are summarized in Table 8-21 on page 8-40.

| *Table 8-21. Workplace API* | |
|---|---|
| **Function** | **Description** |
| WinCreateObject | Create an object. |
| WinDeregisterObjectClass | Deregisters (removes) a Workplace object class. |
| WinDestroyObject | Delete a Workplace object. |
| WinEnumObjectClasses | Get a list of all Workplace classes that have been registered. |
| WinFreeIcon | Free pointer to icon allocated by WinLoadFileIcon |
| WinLoadFileIcon | Get a pointer to an icon associated with the specified icon file. |
| WinQueryObject | Get a handle to a given object. |
| WinRegisterObjectClass | Register a Workplace object class. |
| WinReplaceObjectClass | Replace a registered class with another registered class. |
| WinRestoreWindowPos | Restore a window to the state it was in when WinStoreWindowPos was last called for the same application and key name. |
| WinSetFileIcon | Set the icon for a file. |
| WinSetObjectData | Change settings on an object that was created with the WinCreateObject function. |
| WinShutdownSystem | Close down the system. |
| WinStoreWindowPos | Save the current state of the specified window. |

## Object Class Functions

In order for the Shell to know how to manipulate objects on the user's behalf, the Shell must know about the object class and the class definition, that is, its data and methods. Object classes, therefore, must be registered with the Shell. WinEnumObjectClasses, WinRegisterObjectClass, WinDeRegisterObjectClass, and WinReplaceObjectClass enable applications to affect object classes registered with the Workplace Shell.

WinEnumObjectClasses allows an application to get the list of all Workplace object classes that have been registered with the Shell.

WinRegisterObjectClass and WinDeRegisterObjectClass allow an application to register and deregister Workplace object classes with the Shell. WinRegisterObjectClass registers the specified Workplace object class that is defined in the specified DLL module that was created using the SOM compiler. The name of the object class must match the library name specified in the .DEF file used to build the DLL. Because all registered classes are maintained in the OS2.INI file and are cached upon system initialization, a class should be removed if it is no longer needed.

WinReplaceObjectClass can be used to replace an existing (registered) object class with another class. The replacement class is a subclass of the class being replaced. This type of class is useful for modifying the behavior of instances of a Workplace object class without the instances being aware of the new class. WinReplaceObjectClass can also be used to "undo" the replacement and restore the original class that defines the behavior of its instances.

## Object Instance Functions

Applications can create and destroy object instances of Workplace classes by calling WinCreateObject and WinDestroyObject. To create an object, an application must specify the object's class name, title, setup information, and where it is to be placed on the Desktop.

Each object class defines a set of KEYNAME, or setup, variables and values that can be used to control the attributes and behavior of its objects. An object's setup information is processed by its wpSetup method inherited from WPObject. The object's wpSetup method is called when the object is created by WinCreateObject. Every Workplace class inherits the set of KEYNAMES defined for the WPObject class. Every Workplace class can define additional KEYNAMES, unique to that class. KEYNAMES defined by the Workplace objects provided with the Shell are shown in Table 8-17 on page 8-32.

Newly-created objects need to be placed somewhere on the Desktop, either directly on the Desktop or in a folder. All Workplace objects have object IDs associated with them. As shown in Table 8-22, predefined system folders have object IDs associated with them. When an object is created by WinCreateObject, an application can specify its location as one of the predefined system folders.

*Table 8-22. Object IDs for Predefined System Folders*

| ID | System Folder |
|---|---|
| <WP_NOWHERE> | The hidden folder. |
| <WP_DESKTOP> | The Desktop. |
| <WP_SYSTEM> | The System folder. |
| <WP_TEMPS> | The Templates folder. |
| <WP_CONFIG> | The System Setup folder. |
| <WP_START> | The Startup folder. |
| <WP_INFO> | The information folder. |
| <WP_DRIVE> | The Drives folder. |

WinCreateObject returns a handle to the newly-created object. This handle is persistent and can be used at any time to reference the object. Applications that did not create the object, can get a handle to the object by calling WinQueryObject. WinQueryObject requires either a full path name for a file object or an objectID.

Applications that can get a handle to an object can change the behavior or state of that object by calling WinSetObjectData. This function results in the Workplace Shell calling the object's wpSetup method. By specifying a value for KEYNAME variables defined for the object's class, an application can, therefore, effect changes to objects that already exist on the Desktop. The process of effecting changes to existing objects on the Desktop is summarized in Figure 8-28.

```
WinQueryObject(...); /* use ObjectID to get handle to object          */
pszSetupString="KEYNAME1=value;...";/*Put KEYNAME values in Setup String*/
WinSetObjectData(...);    /*Change behavior of existing object        */
```

*Figure 8-28. Changing Existing Objects on the Desktop*

## REXX Utility Workplace Functions

The REXX Language provides utility functions for Workplace classes and objects. These functions are listed in Table 8-23.

*Table 8-23. REXX Utility Workplace Functions*

| Function | Description |
|---|---|
| SysRegisterObjectClass | Register a Workplace object class |
| SysDeregisterObjectClass | Deregister a Workplace object class |
| SysQueryClassList | Get the list of Workplace class registered with the Shell. |
| SysCreateObject | Create a Workplace object. |

REXX utility functions allow users to write batch files to operate on Workplace classes and objects in the same way as applications. For example, a simple REXX batch file, as shown in Figure 8-29, can list all Workplace classes registered with the Shell.

```
/* Load the REXX utility functions                          */
call RxFuncAdd 'SysLoadFuncs', 'RexxUtil', 'SysLoadFuncs'
call SysLoadFuncs
/* List all Workplace classes registered with the Shell */
call SysQueryClassList   "list."
do i = 1 to list.0
  say 'Class' i 'is' list.i
end
```

*Figure 8-29. REXX Batch File to List all Classes Registered With Workplace*

## Installing a Workplace Object

Workplace objects can be installed on the Desktop in one of two ways:

- By running an installation program or batch file
- By using the Workplace Class List utility

### Object Installation Programs

Application developers can provide installation programs for their objects. From the user's perspective, installation consists of putting a diskette in the diskette drive, double-clicking on the diskette drive and then on the installation program.

An installation program is responsible for:

- Copying the DLL that contains the object's class definition from a diskette to the \OS2\DLL directory or to a directory in the LIBPATH.
- Registering the class and its DLL name with the Shell by calling WinRegisterObjectClass.
- Creating an object instance of the class and placing it on the Desktop or in a particular folder by calling WinCreateObject.

An example of an installation program for a Workplace object is shown in Figure 8-30 on page 8-43.

```
/*  Command-line program to install workplace objects        */

#define INCL_WINWORKPLACE
#include <os2.h>
#include <stdio.h>
#include <string.h>

#if defined(DEBUG)
   #define LOCATION_DESKTOP ((PSZ)"<WP_DESKTOP>")
#endif

/************************************************************/
*  Main Function
*
*   argv[1] = Class Name
*   argv[2] = Module (DLL) Name
*   argv[3] = Object Title
*   argv[4] = Location
*   argv[5] = Setup String
*
************************************************************/

INT main (argc, argv)
   INT argc;
   CHAR *argv[];
{
HAB vhab;
HMQ vhmq;
BOOL fSuccess;

   if (argc == 1)
   {
      #if defined(DEBUG)
      {
         printf("Usage:\n\n");
         printf("   WPCREATE ClassName ModuleName Title [[Location]
                [SetupString]]\n");
      }
      #endif
      return (0);
   }   /* end if (argc == 1) */

   if (argc < 4) return (1);        /* first three parms are mandatory */

   /*   Register the class              */

   #if defined(DEBUG)
      printf("WinRegisterObjectClass(%s, %s)...\n", argv[1], argv[2]);
   #endif
```

*Figure 8-30 (Part 1 of 2). Installation Program for Workplace Object*

```
    fSuccess =
    WinRegisterObjectClass(
       argv[1],                              /* Class name (case sensitive) */
       argv[2]);                                          /* module name */

    if (!fSuccess)          return (1);       /* return non-zero for error */

    #if defined(DEBUG)
       printf("Success: rc = %u\n", fSuccess);
    #endif

    /*   Create an instance of the object     */

    #if defined(DEBUG)
       printf("WinCreateObject(%s, %s,...)...\n", argv[1], argv[3]);
    #endif
    fSuccess =
    WinCreateObject(
       argv[1],                                                /* class name */
       argv[3],                                      /* object name (Title) */
       argc > 5 ? argv[5] : " ",                                /* setup string */
       argc > 4 ? argv[4] : LOCATION_DESKTOP,                     /* location */
       CO_FAILIFEXISTS);                                            /* flags */
    if (!fSuccess)  return (1);                  /* return non-zero for error */

    #if defined(DEBUG)
       printf("Success: rc = %u\n", fSuccess);
    #endif

    return(0);

}   /* end main() */
```

*Figure 8-30 (Part 2 of 2). Installation Program for Workplace Object*

Instantiating an object is an optional responsibility for an installation program. When a class is registered by calling WinRegisterObjectClass, an object template is placed in the Templates Folder on the Desktop, if the class supports templating. Users can create instances of these objects by "tearing off" a copy of the template. This can be useful for larger applications that define data file objects that are associated with program objects.

## Object Installation Batch Files

An installation batch file written in the REXX language performs the same operations, but uses the REXX-language utility functions SysRegisterObjectClass, SysCreateObject, SysDeRegisterObjectClass instead of the WinRegisterObjectClass, WinCreateObject and WinDeRegisterObjectClass Workplace functions. An example of an installation batch file written using the REXX-language utility functions is shown in Figure 8-31 on page 8-45. For more information, see *Procedures Language 2/REXX Reference* in the OS/2 2.0 Technical Library.

```
/* Register a Workplace class and create an instance     */

/* Load the REXX utility functions                       */
call RxFuncAdd 'SysLoadFuncs', 'RexxUtil', 'SysLoadFuncs'
call SysLoadFuncs

/* Register class with Shell                             */
if SysRegisterObjectClass("NewObjectClass", "NEWDLL") then
   say 'Class Registration successfully completed for NewObject'

/* Create instance of object                             */
if SysCreateObject("NewObjectClass","A New Object",
                   "<WP_Desktop>", "OBJECTID=<A New Object>") then
   say 'A New Object successfully created and placed on Desktop'
```

*Figure 8-31. REXX Batch File for Installing Workplace Objects*

## The Workplace Class List Object

The Toolkit provides a Workplace Class List Object that is automatically installed during installation of the Toolkit. This object is a tool that provides a windowed user interface for general Workplace class registration and object creation activities. It performs all the functions that a typical Workplace object installation program must provide, with the exception of copying files from an installation diskette to a hard disk. It is a fast and easy way to build and test objects in an application development environment. It is not a tool for the general OS/2 user, who knows nothing about Workplace classes but only about Workplace objects.

The Workplace Class List Object displays a hierarchical list of all classes registered with the Shell. The user can add a class to this list or perform a number of actions on a specific class in the list. The user can create an instance of the class, replace and unreplace the class, and delete the class.

The Workplace Class List uses WinEnumObjectClasses to get the list of all classes registered with the Shell. WinEnumObjectClasses returns only the name of the class and the DLL module that contains the class definition. It does not return information on the class ancestry that can be used to construct a hierarchical list of classes. Because the Workplace Class List is a Workplace object, however, it can call the somParent function to determine parentage of each class and use this information to construct the list.

The Workplace Class List uses the WinRegisterObjectClass, WinDeregisterObjectClass, WinReplaceObjectClass, and WinCreateObject functions to register, deregister, and replace object classes and to create objects.

**Usage Note**

This tool can be used to delete any Workplace class other than the predefined Workplace classes provided with OS/2 2.0. Its users should understand Workplace classes and how they are defined so that they can be recovered. Because the general user may not have this level of understanding, application developers should not distribute this tool with their objects to their customers. The recommended way of delivering objects to customers is through Installation Programs or Batch Files.

## Programming Considerations for the Workplace

Applications developers should be aware of the following items with respect to the Workplace.

**Extended Attributes and the Workplace**

Applications not written for the Workplace need to be aware of how .LONGNAME and .ASSOCTABLE extended attributes are used by the Workplace.

In the Workplace, the user can edit the name of an object. When an object is a file system object, the Workplace renames the file object to match the name the user has entered. When the file system object resides on an HPFS disk, the new file name can have a long name and can accommodate whatever the user has entered. When the file system object resides on a FAT disk, the new file name must be no longer than eight characters. If the user has entered a name longer than eight characters, the Workplace uses the first eight characters to rename the object and places the remaining characters in a .LONGNAME extended attribute associated with the file system object. This means that the title of a file object is the .LONGNAME extended attribute or the file name, if no .LONGNAME exists.

An .ASSOCTABLE extended attribute contains information that associates data files with the applications that create them or that know how to use them. Applications that a data file has been associated with appear in the list of **Open** actions for the data file. This means that "opening the file" is equivalent to starting the application that creates or modifies that file. The application is passed the name of the file as a command-line parameter.

.ASSOCTABLE extended attributes are defined in an application's resource file as shown in Figure 8-32.

```
ASSOCTABLE
BEGIN
  [file type], [file extension], [flags], [icon file name]
END
```

*Figure 8-32. .ASSOCTABLE Extended Attributes*

When an application that defines an .ASSOCTABLE is installed in the Workplace, the Shell automatically creates object templates for each type of data file that has been associated with the application.

For more information on extended attributes, see the *OS/2 Programming Guide, Volume I* in the OS/2 2.0 Technical Library.

**Printing in the Workplace Shell**

In order to support printing the contents of an object from the Desktop—by using the object's pop-up menus or by dragging the object and dropping it on the printer object—the object's class definition must override the wpPrintObject method that is inherited from its parent class.

The new class is created as a subclass of an existing class. For example, if the object is to be a data file, the wpDataFile class would be used as the parent class. In the class definition file, the new class overrides the wpPrintObject method that it inherits from its parent class. The new class's version of wpPrintObject contains the code that prints the contents of the object.

**Note:** It is recommended that the code in the object's version of wpPrintObject start a separate thread to execute the code that actually prints the contents of the object. By doing the printing in a separate thread, control can be returned to the Shell and the user immediately. Supporting code, such as dialog windows, should be in the separate thread as well.

If this object class is to recognize files that already exist, override the wpclsQueryInstanceType and the wpclsQueryInstanceFilter class methods to check for appropriate .TYPE EAs or file extensions.

## Summary

*Table 8-24 (Page 1 of 2). Summary of Workplace Terms*

| Term | Description |
|---|---|
| **Conditional Cascaded Menu** | A pull-down menu associated with an item that has a cascade mini-push button beside it in an object's pop-up menu. The pull-down menu is displayed when the user selects the mini-push button. |
| **Class** | A general description of an object. Classes define data which represents the state of an object and methods that define the object's behavior and change the state of the object. |
| **Direct Manipulation** | The user's ability to interact with an object by using the mouse, typically by dragging an object around on the desktop and dropping it on other objects. |
| **Encapsulation** | Hiding an object's implementation, that is, its private, internal data and methods. Private variables and methods are accessible only to the object that contains them. |
| **Extended Attribute** | Contain information about a file object. The information contained in EAs is more comprehensive and varied than the information displayed by the DIR command. EAs are not part of the file object and are maintained by the file system |
| **Hierarchical Inheritance** | The relationship between parent and child classes. Inheritance between objects is hierarchical. An object that is lower in the inheritance hierarchy than another object (that is, a child object of a parent or ancestor object) inherits all the characteristics and behaviors of the objects above it in the hierarchy (ancestor objects). |
| **Inheritance** | The derivation of new (child) classes from existing (parent) classes. The new class inherits all the data and methods of the parent class without having to redefine them. |
| **Metaclass** | Defines class methods and class properties for the class of a class. Class methods act on class data common to all object instances of the class. |
| **Method** | A method is a function that defines a behavior for a class or object. |
| **Object** | A specific instance, or instantiation, of a class. Objects have associated data, called the object's state, and a set of behaviors, called the object's methods. |
| **Polymorphism** | The ability to have different implementations of the same method for two or more classes of objects. |
| **Pop-up menu** | A menu that lists the actions that a user can perform on an object. The contents of the pop-up menu can vary depending on the context, or state, of the object. |
| **Persistent object** | An object whose instance data and state are preserved between system shutdown and system startup. |
| **Settings Notebook** | A control window that is used to display the settings for an object and to enable the user to change them. |

*Table 8-24 (Page 2 of 2). Summary of Workplace Terms*

| Term | Description |
|---|---|
| **Shadow** | An object that refers to another object's data. A shadow is not a copy of another object, but is another representation of the object. |
| **Subclass** | A class that inherits the same characteristics and behaviors of its parent, and has its own characteristics and behaviors. |
| **System Object Model (SOM)** | A mechanism that enables programmers to use object-oriented programming techniques in a language-independent manner. |
| **Template** | The primary user mechanism for creating new instances of a Workplace object. |
| **View** | A way of looking at an object's information. |
| **Workplace Shell** | The OS/2 2.0 object-oriented, graphical user interface. |

# Appendix A. Sample Programs Cross Index

The Toolkit provides sample programs that demonstrate how to use the OS/2 API functions. The files that make up each sample program are located in specific sample program directories. The following tables shows the API functions, and the sample program directories (and thereby the sample programs) that demonstrate their use. The API functions are broken down according the following groups:

- Control program functions
- Device functions
- Direct manipulation functions
- Graphics programming interface functions
- Profile functions
- Window functions

## Control Program Functions

The following table shows all the Control Program (Dos) functions in alphabetic order.

| Function Name | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DosAllocMem | | | | √ | | | | | √ | | √ | √ | | √ | | √ | | | | √ | | | √ | | | | | √ | √ | |
| DosAllocSharedMem | | | | | √ | | | | | | | | | √ | √ | | | | | | | | | | | | | | | |
| DosAsyncTimer | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| DosBeep | | | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | | √ | | | | | | √ | | √ | √ | √ | | | √ | √ | √ |
| DosClose | | | | √ | | √ | | | √ | | | √ | √ | √ | √ | √ | | | | √ | √ | | √ | √ | | | | | | |
| DosCloseEventSem | | | | | | √ | | | | | | | √ | | √ | √ | | | | | √ | | | | | | | | | |
| DosCloseMutexSem | | | | | | √ | | | | | | | | √ | | √ | | | | | √ | | | | | | | | | |
| DosCloseMuxWaitSem | | | | | | | | | | | | | | | | | | | | | √ | | | | | | | | | |
| DosCloseQueue | | | | | | | | | | | | | | | √ | | | | | | | | | | | | | | | |
| DosConnectNPipe | | | | | | | | | | | | | √ | | | | | | | | | | | | | | | | | |
| DosCopy | | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | |
| DosCreateEventSem | | | | | | √ | √ | | | | √ | | √ | √ | √ | √ | | | | | √ | | | √ | √ | | | | | |
| DosCreateMutexSem | | | | | | √ | | | | | √ | | | √ | | √ | | | | | √ | | | | | | | | | |
| DosCreateMuxWaitSem | | | | | | | | | | | | | | | | | | | | | √ | | | | | | | | | |
| DosCreateNPipe | | | | | | | | | | | | | √ | | | | | | | | | | | | | | | | | |
| DosCreatePipe | | | | | | | | | | | | | | | | √ | | | | | | | | | | | | | | |
| DosCreateQueue | | | | | | | | | | | | | | | √ | | | | | | | | | | | | | | | |
| DosCreateThread | | | | | √ | √ | √ | | | | √ | | | √ | | √ | | | | | √ | | | √ | | | | | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | ANIMALS | 9 | IMAGE | 17 | REXX\REXXSAMP\CALLREXX | 25 | CLOCK |
| 2 | CLIPBRD | 10 | IPF | 18 | REXX\REXXSAMP\DEVINFO | 26 | TP |
| 3 | DIALOG | 11 | JIGSAW | 19 | REXX\REXXSAMP\RXMACDLL | 27 | VDD |
| 4 | DDLAPI | 12 | EAS | 20 | REXX\REXXSAMP\REXXUTIL | 28 | VMM |
| 5 | DRAGDROP | 13 | NPIPE | 21 | SEMAPH | 29 | WORMS |
| 6 | GRAPHIC | 14 | PRINT | 22 | SORT | 30 | WPCAR |
| 7 | HANOI | 15 | QUEUES | 23 | STYLE | | |
| 8 | HELLO | 16 | REXX\REXXSAMP\PMREXX | 24 | TEMPLATE | | |

| Function Name | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DosDelete | | | | | √ | | | | | | | | | | | | | | | √ | | | | | | | | | | |
| DosDevConfig | | | | | | | | | | | | | | | | | | √ | | | | | | | | | | | | |
| DosDisConnectNPipe | | | | | | | | | | | | | √ | | | | | | | | | | | | | | | | | |
| DosDupHandle | | | | | | | | | | | | | | | | √ | | | | | | | | | | | | | | |
| DosEnterCritSec | | | | | √ | | √ | | | | | | √ | | | √ | | | | | | √ | | | | | | | | |
| DosEnumAttribute | | | | | | | | | | | | √ | | | | | | | | | | | | | | | | | | |
| DosErrClass | | | | | | | | | | | | | | | | √ | | | | | | | | | | | | | | |
| DosError | | | | | | | | | | | | | | | | √ | | | | √ | | | | | | | | √ | | |
| DosExecPgm | | | | | | | | | | | | | √ | | √ | | | | | | | | | | | | | | | |
| DosExit | | √ | | √ | | √ | √ | | √ | | √ | √ | √ | √ | | √ | | | | | √ | √ | √ | √ | | | | √ | √ | |
| DosExitCritSec | | | | | | | √ | | | | | | √ | | | √ | | | | | | √ | | | | | | | | |
| DosExitList | | | | | | | √ | | | | | | √ | | | √ | | | | | | √ | | | | | | | | |
| DosFindClose | | | | | √ | | | | | | | | | | | | | | | √ | | | | | | | | | | |
| DosFindFirst | | | | √ | √ | | | | √ | | √ | √ | | | | | | | | √ | | | | | | | | | | |
| DosFindNext | | | | √ | √ | | | | | | | √ | | | | | | | | √ | | | | | | | | | | |
| DosFreeMem | | | | √ | | | | | √ | | √ | √ | | √ | √ | √ | √ | | | √ | | | √ | | | | | √ | √ | |
| DosFreeModule | | | | | | | | | | √ | | | | | | | | | | | | | | | | | | | | |
| DosGetDateTime | | | | | | | | | | | √ | | | | | | | | | √ | | | | | √ | | | | | |
| DosGetInfoBlocks | √ | | | | | √ | | | | | √ | | √ | √ | √ | √ | | | | | | | | | √ | √ | | | | √ |
| DosGetMessage | | | | | | | | | | | | | | | | √ | | | | √ | | | | | | | | | | |
| DosGetNamedSharedMem | | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | |
| DosGiveSharedMem | | | | | | | | | | | | | | | √ | | | | | | | | | | | | | | | |
| DosLoadModule | | | | | | | | | | √ | | | | | | | | | | | | | | | | | | | | √ |
| DosMove | | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | |
| DosOpen | | | | √ | | | | | √ | | √ | √ | √ | √ | √ | √ | | | | √ | | | √ | √ | | | | | | |
| DosOpenQueue | | | | | | | | | | | | | | | √ | | | | | | | | | | | | | | | |
| DosPostEventSem | | | | | | √ | √ | | | | √ | | √ | √ | √ | √ | | | | | √ | | | √ | √ | | | | | |
| DosQueryCtryInfo | | | | | | | | | | | | | | | | √ | | | √ | | | | | | √ | | | | | |
| DosQueryCurrentDir | | | | √ | √ | | | | | | | √ | | | | | | | | | | | | | | | | | | |
| DosQueryCurrentDisk | | | | √ | √ | | | | | | | √ | | | | | | | | √ | | | | | | | | | | |
| DosQueryEventSem | | | | | | | | | | | √ | | | | √ | | | | | | | | | | | | | | | |
| DosQueryFHState | | | | | | | | | | | | | | | | √ | | | | | | | | | | | | | | |
| DosQueryFileInfo | | | | | | | | | | | √ | | | √ | | | | | | √ | | | √ | | | | | | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | ANIMALS | 9 | IMAGE | 17 | REXX\REXXSAMP\CALLREXX | 25 | CLOCK |
| 2 | CLIPBRD | 10 | IPF | 18 | REXX\REXXSAMP\DEVINFO | 26 | TP |
| 3 | DIALOG | 11 | JIGSAW | 19 | REXX\REXXSAMP\RXMACDLL | 27 | VDD |
| 4 | DDLAPI | 12 | EAS | 20 | REXX\REXXSAMP\REXXUTIL | 28 | VMM |
| 5 | DRAGDROP | 13 | NPIPE | 21 | SEMAPH | 29 | WORMS |
| 6 | GRAPHIC | 14 | PRINT | 22 | SORT | 30 | WPCAR |
| 7 | HANOI | 15 | QUEUES | 23 | STYLE | | |
| 8 | HELLO | 16 | REXX\REXXSAMP\PMREXX | 24 | TEMPLATE | | |

| Function Name | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DosQueryFSAttach | | | | | | | | | | | | | | | | | | | | √ | | | | | | | | | | |
| DosQueryFSInfo | | | | | | | | | | | | | | | | | | | | √ | | | | | | | | | | |
| DosQueryMem | | | | | | | | | | | | | | | | | | | | | | | | | | | | √ | | |
| DosQueryPathInfo | | | | | | | | | | | | √ | √ | | | | | | | √ | | | | | | | | | | |
| DosQuerySysInfo | | | | | | | | | | | | | | | | | | | | √ | | | | | | | | | | |
| DosRead | | | | √ | | | | | √ | | √ | | √ | √ | | √ | | | | | | | √ | | | | | | | |
| DosReadQueue | | | | | | | | | | | | | | | √ | | | | | | | | | | | | | | | |
| DosReleaseMutexSem | | | | | | √ | | | | | √ | | | √ | | √ | | | | | √ | | | | | | | | | |
| DosRequestMutexSem | | | | | | √ | | | | | √ | | | √ | | √ | | | | | √ | | | | | | | | | |
| DosResetEventSem | | | | | | √ | | | | | √ | | √ | √ | √ | √ | | | | | √ | | | | √ | | | | | |
| DosResumeThread | | | | | | | | | | | | | | | | | | | | | √ | | | | | | | | √ | |
| DosSearchPath | | | | | | | | | | | | | | | | √ | | | | √ | | | | | | | | | | |
| DosSetCurrentDir | | | | √ | √ | | | | | | | √ | | | | | | | | | | | | | | | | | | |
| DosSetDateTime | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| DosSetDefaultDisk | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | | |
| DosSetExceptionHandler | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | √ | | |
| DosSetFHState | | | | | | | | | | | | | | | | √ | | | | | | | | | | | | | | |
| DosSetFileInfo | | | | | | | | | | | | | | | | | | | | √ | | | | | | | | | | |
| DosSetMem | | | | √ | | | | | | | | √ | | | | | | | | | | | | | | | | √ | | |
| DosSetPathInfo | | | | | | | | | | | | √ | | | | | | | | | | | | | | | | | | |
| DosSetPriority | | | | | | √ | | | | | √ | | | √ | | | | | | | | √ | | | | | | | | |
| DosSleep | | | | | | | | | | | | | √ | | √ | √ | | | | √ | | √ | | | | | | | √ | √ |
| DosStartTimer | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| DosStopTimer | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| DosSubAllocMem | | | | | √ | | | | | | | | | | √ | | | | | | | | | | | | | | | |
| DosSubFreeMem | | | | | √ | | | | | | | | | | √ | | | | | | | | | | | | | | | |
| DosSubSetMem | | | | | √ | | | | | | | | | | √ | | | | | | | | | | | | | | | |
| DosSuspendThread | | | | | | | | | | | √ | | | | | √ | | | | | | | | | | | | | √ | |
| DosUnsetExceptionHandler | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | √ | | |
| DosWaitEventSem | | | | | | √ | √ | | | | | | √ | √ | √ | √ | | | | | √ | | | √ | √ | | | | | |
| DosWaitMuxWaitSem | | | | | | | | | | | | | | | | | | | | | √ | | | | | | | | | |
| DosWaitNPipe | | | | | | | | | | | | | √ | | | | | | | | | | | | | | | | | |
| DosWaitThread | | | | | | √ | √ | | | | | | | √ | | | | | | | √ | | | √ | | | | | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | ANIMALS | 9 | IMAGE | 17 | REXX\REXXSAMP\CALLREXX | 25 | CLOCK |
| 2 | CLIPBRD | 10 | IPF | 18 | REXX\REXXSAMP\DEVINFO | 26 | TP |
| 3 | DIALOG | 11 | JIGSAW | 19 | REXX\REXXSAMP\RXMACDLL | 27 | VDD |
| 4 | DDLAPI | 12 | EAS | 20 | REXX\REXXSAMP\REXXUTIL | 28 | VMM |
| 5 | DRAGDROP | 13 | NPIPE | 21 | SEMAPH | 29 | WORMS |
| 6 | GRAPHIC | 14 | PRINT | 22 | SORT | 30 | WPCAR |
| 7 | HANOI | 15 | QUEUES | 23 | STYLE | | |
| 8 | HELLO | 16 | REXX\REXXSAMP\PMREXX | 24 | TEMPLATE | | |

| Function Name | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DosWrite | | | | | | | | | | | | | √ | | | √ | | | | | | | √ | | | | | | | |
| DosWriteQueue | | | | | | | | | | | | | | | √ | | | | | | | | | | | | | | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | ANIMALS | 9 | IMAGE | 17 | REXX\REXXSAMP\CALLREXX | 25 | CLOCK |
| 2 | CLIPBRD | 10 | IPF | 18 | REXX\REXXSAMP\DEVINFO | 26 | TP |
| 3 | DIALOG | 11 | JIGSAW | 19 | REXX\REXXSAMP\RXMACDLL | 27 | VDD |
| 4 | DDLAPI | 12 | EAS | 20 | REXX\REXXSAMP\REXXUTIL | 28 | VMM |
| 5 | DRAGDROP | 13 | NPIPE | 21 | SEMAPH | 29 | WORMS |
| 6 | GRAPHIC | 14 | PRINT | 22 | SORT | 30 | WPCAR |
| 7 | HANOI | 15 | QUEUES | 23 | STYLE | | |
| 8 | HELLO | 16 | REXX\REXXSAMP\PMREXX | 24 | TEMPLATE | | |

## Device Functions

The following table shows all the Device (Dev) functions in alphabetic order.

| Function Name | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DevCloseDC | | √ | | | | √ | | | √ | | √ | | | √ | | | | | | | | | | √ | √ | | | | | |
| DevEscape | | | | | | √ | | | | | | | | √ | | | | | | | | | | √ | | | | | | |
| DevOpenDC | | √ | | | | √ | | | √ | | √ | | | √ | | | | | | | | | | √ | √ | | | | | |
| DevPostDeviceModes | | | | | | √ | | | | | | | | √ | | | | | | | | | | | | | | | | |
| DevQueryCaps | | | | | | √ | | | | | | | | √ | | √ | | | | | | | | | √ | | | | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | ANIMALS | 9 | IMAGE | 17 | REXX\REXXSAMP\CALLREXX | 25 | CLOCK |
| 2 | CLIPBRD | 10 | IPF | 18 | REXX\REXXSAMP\DEVINFO | 26 | TP |
| 3 | DIALOG | 11 | JIGSAW | 19 | REXX\REXXSAMP\RXMACDLL | 27 | VDD |
| 4 | DDLAPI | 12 | EAS | 20 | REXX\REXXSAMP\REXXUTIL | 28 | VMM |
| 5 | DRAGDROP | 13 | NPIPE | 21 | SEMAPH | 29 | WORMS |
| 6 | GRAPHIC | 14 | PRINT | 22 | SORT | 30 | WPCAR |
| 7 | HANOI | 15 | QUEUES | 23 | STYLE | | |
| 8 | HELLO | 16 | REXX\REXXSAMP\PMREXX | 24 | TEMPLATE | | |

## Direct Manipulation Functions

This section describes functions that an application would use to initiate or participate in a direct manipulation operation. The following table shows all the direct manipulation (Drg) functions in alphabetic order.

| Function Name | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DrgAccessDraginfo | | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | |
| DrgAddStrHandle | | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | |
| DrgAllocDraginfo | | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | |
| DrgAllocDragtransfer | | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | |
| DrgDeleteDraginfoStrHandles | | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | |
| DrgDrag | | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | |
| DrgFreeDraginfo | | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | |
| DrgFreeDragtransfer | | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | |
| DrgQueryDragitem | | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | |
| DrgQueryDragitemCount | | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | |
| DrgQueryDragitemPtr | | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | |
| DrgQueryStrName | | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | |
| DrgSendTransferMsg | | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | |
| DrgSetDragItem | | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | |
| DrgVerifyRMF | | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | ANIMALS | 9 | IMAGE | 17 | REXX\REXXSAMP\CALLREXX | 25 | CLOCK | |
| 2 | CLIPBRD | 10 | IPF | 18 | REXX\REXXSAMP\DEVINFO | 26 | TP | |
| 3 | DIALOG | 11 | JIGSAW | 19 | REXX\REXXSAMP\RXMACDLL | 27 | VDD | |
| 4 | DDLAPI | 12 | EAS | 20 | REXX\REXXSAMP\REXXUTIL | 28 | VMM | |
| 5 | DRAGDROP | 13 | NPIPE | 21 | SEMAPH | 29 | WORMS | |
| 6 | GRAPHIC | 14 | PRINT | 22 | SORT | 30 | WPCAR | |
| 7 | HANOI | 15 | QUEUES | 23 | STYLE | | | |
| 8 | HELLO | 16 | REXX\REXXSAMP\PMREXX | 24 | TEMPLATE | | | |

## GPI Functions by Functional Area

The following table shows how all of the Graphics Programming Interface (GPI) functions are related within functional areas. The functions are in alphabetic order within these areas.

| Function Name | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Curve Functions** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Primitive Functions | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiFullArc | | | | | | √ | | | | | | | | √ | | | | | | | | | | | √ | | | | | |
| **Bit-Map Support** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Creation and Selection Functions | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiCreateBitmap | | √ | | | | | | | √ | | √ | | | √ | | | | | | | | | | | √ | | | | | |
| GpiDeleteBitmap | | √ | | | | | | | √ | √ | √ | | | √ | | | | | | | | | | | √ | | | | | |
| GpiLoadBitmap | | √ | | | | | | | | √ | | √ | | | | | | | | | | | | | | | | | | |
| GpiSetBitmap | | √ | | | | | | | √ | | √ | | | √ | | | | | | | | | | | √ | | | | | |
| Operations on Raw Bit Maps | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiQueryBitmapInfoHeader | | | | | | | | | | | | | | √ | | | | | | | | | | | | | | | | |
| GpiQueryBitmapParameters | | √ | | | | | | | | | | | | √ | | | | | | | | | | | | | | | | |
| GpiQueryDeviceBitmapFormats | | | | | | | | | | | | | | √ | | | | | | | | | | | | | | | | |
| GpiSetBitmapBits | | | | | | | | | | | √ | | | √ | | | | | | | | | | | | | | | | |
| Operations through Presentation Spaces | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiBitBlt | | √ | | | | | | | | | √ | | | √ | | | | | | | | | | | √ | | | | | |
| GpiQueryPel | | | | | | | | | | | √ | | | | | | | | | | | | | | | | | | | |
| **Character Functions** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Attribute Setting Functions | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiQueryCharBox | | | | | | √ | | | | | | | | √ | | | | | | | | | | | √ | | | | | |
| GpiSetCharBox | | | | | | | | | | | | | | √ | | | | | | | | | | | √ | | | | | |
| GpiSetCharMode | | | | | | | | | | | | | | √ | | | | | | | | | | | √ | | | | | |
| GpiSetCharSet | | | | | | | | | | | | | | √ | | | | | | | | | √ | | | | | | | |
| Primitive Functions | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiCharString | | | | | | | | | | | | | | √ | | | | | | | | | | | √ | | | | | |
| GpiCharStringAt | | | √ | √ | | √ | √ | √ | | | | √ | √ | √ | √ | √ | | | | | | √ | | | | | | √ | | |
| GpiCharStringPosAt | | | | | | | | | | | | | | | | | | | | | √ | | | | | | | | | |
| GpiQueryCharStringPos | | | | | | | | | | | | | | √ | | | | | | | | | | | | | | | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | ANIMALS | 9 | IMAGE | 17 | REXX\REXXSAMP\CALLREXX | 25 | CLOCK |
| 2 | CLIPBRD | 10 | IPF | 18 | REXX\REXXSAMP\DEVINFO | 26 | TP |
| 3 | DIALOG | 11 | JIGSAW | 19 | REXX\REXXSAMP\RXMACDLL | 27 | VDD |
| 4 | DDLAPI | 12 | EAS | 20 | REXX\REXXSAMP\REXXUTIL | 28 | VMM |
| 5 | DRAGDROP | 13 | NPIPE | 21 | SEMAPH | 29 | WORMS |
| 6 | GRAPHIC | 14 | PRINT | 22 | SORT | 30 | WPCAR |
| 7 | HANOI | 15 | QUEUES | 23 | STYLE | | |
| 8 | HELLO | 16 | REXX\REXXSAMP\PMREXX | 24 | TEMPLATE | | |

| Function Name | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Resources and Defaults Functions** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiCreateLogFont | | | | | | | | | | | | | | √ | | √ | | | | | | | √ | | | | | | | |
| GpiDeleteSetId | | | | | | | | | | | | | | √ | | | | | | | | | √ | | | | | | | |
| GpiQueryCp | | | | | | | | | | | | | | √ | | | | | | | | | | | | | | | | |
| GpiQueryFontMetrics | | | | | | √ | | | | | | | √ | √ | √ | √ | | | | | √ | | √ | | | | | | | |
| GpiQueryFonts | | | | | | | | | | | | | | | | √ | | | | | | | √ | | | | | | | |
| **Color and Mix Functions** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Attribute Setting Functions** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiQueryColor | | √ | | | | | | | | | | | | | | | | | | | √ | | | | √ | | | | | |
| GpiSetBackColor | | | | | | | | √ | √ | | | | | | | | | | | | √ | | | | | | | | | |
| GpiSetBackMix | | | | | | | | √ | √ | | | | | √ | | | | | | | | | | | | | | | | |
| GpiSetColor | | √ | √ | | | √ | | √ | √ | | √ | | | √ | | | | | | | √ | √ | | | √ | | | √ | | |
| GpiSetMix | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| **Resources and Default Functions** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiCreateLogColorTable | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| GpiQueryColorIndex | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| GpiQueryRGBColor | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| **Control Functions** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiAssociate | | √ | | | | √ | | | √ | | √ | | | √ | | | | | | | | | | √ | | | | | | |
| GpiCreatePS | | √ | | | | √ | | | √ | | √ | | | √ | | | | | | | | | | √ | √ | | | | | |
| GpiDestroyPS | | √ | | | | √ | | | √ | | √ | | | √ | | | | | | | | | | √ | √ | | | | | |
| GpiQueryDevice | | | | | | | | | | | | | | √ | | √ | | | | | | | | | | | | | | |
| GpiResetPS | | | | | | √ | | | | | | | | √ | | | | | | | | | | | √ | | | | | |
| GpiRestorePS | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| GpiSavePS | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| GpiSetPS | | | | | | | | | | | | | | | | | | | | | | | | | | | | √ | | |
| **Correlation and Boundary Determination Functions** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Bounds Data Functions** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiQueryBoundaryData | | | | | | √ | | | | | | | | √ | | | | | | | | | | | | | | | | |
| GpiResetBoundaryData | | | | | | √ | | | | | | | | √ | | | | | | | | | | | | | | | | |
| **Drawing Functions** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiErase | | | | | √ | √ | | | √ | | | | √ | √ | √ | | | | | | √ | | | | | | | | | |
| GpiQueryStopDraw | | | | | | √ | | | | | | | | √ | | | | | | | | | | | | | | | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | ANIMALS | 9 | IMAGE | 17 | REXX\REXXSAMP\CALLREXX | 25 | CLOCK |
| 2 | CLIPBRD | 10 | IPF | 18 | REXX\REXXSAMP\DEVINFO | 26 | TP |
| 3 | DIALOG | 11 | JIGSAW | 19 | REXX\REXXSAMP\RXMACDLL | 27 | VDD |
| 4 | DDLAPI | 12 | EAS | 20 | REXX\REXXSAMP\REXXUTIL | 28 | VMM |
| 5 | DRAGDROP | 13 | NPIPE | 21 | SEMAPH | 29 | WORMS |
| 6 | GRAPHIC | 14 | PRINT | 22 | SORT | 30 | WPCAR |
| 7 | HANOI | 15 | QUEUES | 23 | STYLE | | |
| 8 | HELLO | 16 | REXX\REXXSAMP\PMREXX | 24 | TEMPLATE | | |

| Function Name | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| GpiSetDrawControl | | | | | | √ | | | | | | | | √ | | | | | | | | | | | | | | | | |
| GpiSetStopDraw | | | | | | √ | | | | | | | | √ | | | | | | | | | | | | | | | | |
| **General Attribute Functions** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Attribute Mode Functions | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiPop | | | | | | √ | | | | | | | | √ | | | | | | | | | | | | | | | | |
| GpiQueryAttrMode | | | | | | √ | | | | | | | | √ | | | | | | | | | | | | | | | | |
| GpiSetAttrMode | | | | | | √ | | | | | √ | | | √ | | | | | | | | | | | | | | | | |
| Attribute Strip Setting Functions | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiSetAttrs | | | | | | √ | | | | | | | | √ | | | | | | | | | | | | | | | | |
| **Image Functions** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Primitive Functions | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiImage | | | | | | | | | √ | | | | | | | | | | | | | | | | | | | | | |
| **Line Functions** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Attribute Setting Functions | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiQueryLineType | | √ | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiSetLineType | | √ | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Primitive Functions | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiBox | | √ | | | | | | | | | | | √ | | | | | | | | | | | | √ | | | √ | | |
| GpiLine | | | | | | | | | | | | | √ | | | | | | | | √ | | | | √ | | | √ | | |
| GpiMove | | | | | | | | | | | √ | | √ | | | | | | | | √ | | | | | | | √ | | |
| GpiPolyLine | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| GpiQueryCurrentPosition | | | | | | | | | √ | | | | | | | √ | | | | | | | | | | | | | | |
| GpiSetCurrentPosition | | √ | | | | √ | | | √ | | | | | √ | | | | | | | | | | | √ | | | | | |
| **Metafile Support** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiCopyMetaFile | | | | | | | | | | | | | | √ | | | | | | | | | | | | | | | | |
| GpiDeleteMetaFile | | | | | | √ | | | | | | | | √ | | | | | | | | | | | | | | | | |
| GpiLoadMetaFile | | | | | | √ | | | | | | | | √ | | | | | | | | | | | | | | | | |
| GpiPlayMetaFile | | | | | | √ | | | | | | | | √ | | | | | | | | | | | | | | | | |
| **Path Functions** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Path Clipping Functions | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiSetClipPath | | | | | | | | | | | √ | | | | | | | | | | | | | | | | | | | |
| Path Definition and Deletion Functions | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiBeginPath | | | | | | | | | | | √ | | | | | | | | | | | | | | √ | | | | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | ANIMALS | 9 | IMAGE | 17 | REXX\REXXSAMP\CALLREXX | 25 | CLOCK |
| 2 | CLIPBRD | 10 | IPF | 18 | REXX\REXXSAMP\DEVINFO | 26 | TP |
| 3 | DIALOG | 11 | JIGSAW | 19 | REXX\REXXSAMP\RXMACDLL | 27 | VDD |
| 4 | DDLAPI | 12 | EAS | 20 | REXX\REXXSAMP\REXXUTIL | 28 | VMM |
| 5 | DRAGDROP | 13 | NPIPE | 21 | SEMAPH | 29 | WORMS |
| 6 | GRAPHIC | 14 | PRINT | 22 | SORT | 30 | WPCAR |
| 7 | HANOI | 15 | QUEUES | 23 | STYLE | | |
| 8 | HELLO | 16 | REXX\REXXSAMP\PMREXX | 24 | TEMPLATE | | |

| Function Name | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| GpiCloseFigure | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| GpiEndPath | | | | | | | | | | | √ | | | | | | | | | | | | | | √ | | | | | |
| **Path Drawing Functions** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiFillPath | | | | | | | | | | | √ | | | | | | | | | | | | | | √ | | | | | |
| **Region Support** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Clipping Region Functions** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiSetClipRegion | | | | | | √ | | | | | √ | | | √ | | | | | | | | | | | | | | | | |
| **Drawing Functions** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiPaintRegion | | | | | | √ | | | | | √ | | | √ | | | | | | | | | | | | | | | | |
| **Region Functions** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiCombineRegion | | | | | | √ | | | | | √ | | | √ | | | | | | | | | | | | | | | | |
| GpiCreateRegion | | | | | | √ | | | | | √ | | | √ | | | | | | | | | | | | | | | | |
| GpiDestroyRegion | | | | | | √ | | | | | √ | | | √ | | | | | | | | | | | | | | | | |
| GpiQueryRegionBox | | | | | | | | | | | √ | | | | | | | | | | | | | | | | | | | |
| GpiSet Region | | | | | | | | | | | √ | | | | | | | | | | | | | | | | | | | |
| **Transform Functions** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Conversion Functions** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiConvert | | | | | | | | | | | √ | | | | | | | | | | | | | | √ | | | √ | | |
| **Device Transforms** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiSetPageViewport | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| **Helper Functions** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiScale | | | | | | √ | | | | | | | | √ | | | | | | | | | | | | | | | | |
| GpiTranslate | | | | | | √ | | | | | | | | √ | | | | | | | | | | | | | | | | |
| **Modelling Transform Functions** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiSetModelTransformMatrix | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| **Viewing Transform Functions** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| GpiQueryDefaultViewMatrix | | | | | | √ | | | | | √ | | | √ | | | | | | | | | | | | | | | | |
| GpiSetDefaultViewMatrix | | | | | | √ | | | | | √ | | | √ | | | | | | | | | | | | | | √ | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | ANIMALS | 9 | IMAGE | 17 | REXX\REXXSAMP\CALLREXX | 25 | CLOCK |
| 2 | CLIPBRD | 10 | IPF | 18 | REXX\REXXSAMP\DEVINFO | 26 | TP |
| 3 | DIALOG | 11 | JIGSAW | 19 | REXX\REXXSAMP\RXMACDLL | 27 | VDD |
| 4 | DDLAPI | 12 | EAS | 20 | REXX\REXXSAMP\REXXUTIL | 28 | VMM |
| 5 | DRAGDROP | 13 | NPIPE | 21 | SEMAPH | 29 | WORMS |
| 6 | GRAPHIC | 14 | PRINT | 22 | SORT | 30 | WPCAR |
| 7 | HANOI | 15 | QUEUES | 23 | STYLE | | |
| 8 | HELLO | 16 | REXX\REXXSAMP\PMREXX | 24 | TEMPLATE | | |

## Profile Functions

The following table shows all the Profile (Prf) functions in alphabetic order.

| Function Name | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| PrfCloseProfile | | | | | | | | | | | | | | √ | | | | | | √ | | | | | | | | | | |
| PrfOpenProfile | | | | | | | | | | | | | | √ | | | | | | √ | | | | | | | | | | |
| PrfQueryProfile | | | | | | | | | | | | | | | | | | | | √ | | | | | | | | | | |
| PrfQueryProfileData | | | | | | | | | | | | | | √ | | √ | | | | √ | | | | | √ | | | | | |
| PrfQueryProfileInt | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| PrfQueryProfileSize | | | | | | √ | | | | | | | | √ | | | | | | | | | | | | | | | | |
| PrfQueryProfileString | | | | | | √ | | | | | | | | √ | | | | | | | | | | | √ | | | | | |
| PrfWriteProfileData | | | | | | | | | | | | | | √ | | √ | | | | √ | | | | | √ | | | | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | ANIMALS | 9 | IMAGE | 17 | REXX\REXXSAMP\CALLREXX | 25 | CLOCK |
| 2 | CLIPBRD | 10 | IPF | 18 | REXX\REXXSAMP\DEVINFO | 26 | TP |
| 3 | DIALOG | 11 | JIGSAW | 19 | REXX\REXXSAMP\RXMACDLL | 27 | VDD |
| 4 | DDLAPI | 12 | EAS | 20 | REXX\REXXSAMP\REXXUTIL | 28 | VMM |
| 5 | DRAGDROP | 13 | NPIPE | 21 | SEMAPH | 29 | WORMS |
| 6 | GRAPHIC | 14 | PRINT | 22 | SORT | 30 | WPCAR |
| 7 | HANOI | 15 | QUEUES | 23 | STYLE | | |
| 8 | HELLO | 16 | REXX\REXXSAMP\PMREXX | 24 | TEMPLATE | | |

## Window Functions by Functional Area

The following table shows how all of the Window (WIN) functions are related within functional areas. The functions are in alphabetic order within these areas.

| Function Name | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Accelerators** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinLoadAccelTable | | | | | | | | | | | | | | | | √ | | | | | | | | | | | | | | |
| WinSetAccelTable | | | | | | | | | | | | | | | | √ | | | | | | | | | | | | | | |
| **Alarms** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinAlarm | | | | √ | √ | | √ | | √ | | √ | √ | √ | | √ | √ | | | | | √ | √ | √ | √ | √ | | | √ | | |
| WinMessageBox | | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | | | | | √ | √ | √ | √ | √ | | | √ | | √ |
| **Clipboard** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinCloseClipbrd | | √ | | | | | | | | | | | | √ | | | | | | | | | √ | | | | | | | |
| WinOpenClipbrd | | √ | | | | | | | | | | | | √ | | | | | | | | | √ | | | | | | | |
| WinQueryClipbrdData | | √ | | | | | | | | | | | | √ | | | | | | | | | | | | | | | | |
| WinQueryClipbrdFmtInfo | | | | | | | | | | | | | | √ | | | | | | | | | √ | | | | | | | |
| WinSetClipbrdData | | √ | | | | | | | | | | | | √ | | | | | | | | | | | | | | | | |
| **Coordinate Mapping** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinMapDlgPoints | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| WinMapWindowPoints | | | | | | √ | | | √ | | √ | | | √ | | | | | | | | | √ | | √ | | | | | |
| **Dialog Boxes** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinDefDlgProc | | √ | √ | √ | √ | √ | √ | | √ | | √ | √ | √ | √ | √ | √ | | | | | √ | √ | √ | √ | √ | | | √ | | √ |
| WinDefFileDlgProc | | | | | | √ | | | √ | | √ | | | √ | | | | | | | | | √ | √ | | | | | | |
| WinDefFontDlgProc | | | | | | | | | | | | | | √ | | | | | | | | | | | | | | | | |
| WinDismissDlg | | | √ | √ | √ | √ | √ | | | | √ | √ | √ | √ | √ | √ | | | | | √ | √ | √ | √ | √ | | | √ | | |
| WinDlgBox | | √ | √ | √ | √ | √ | √ | | √ | | √ | √ | √ | √ | √ | √ | | | | | √ | √ | √ | √ | √ | | | √ | | |
| WinFileDlg | | | | | | √ | | | √ | | √ | | | √ | | √ | | | | | | | √ | √ | | | | | | |
| WinFontDlg | | | | | | | | | | | | | | √ | | √ | | | | | | | √ | | | | | | | |
| WinLoadDlg | | | | | | | | | | | | | | | | √ | | | | | | | | | | | | | | |
| **Drawing Management** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Drawing | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinBeginPaint | | | √ | √ | √ | √ | √ | √ | √ | √ | | √ | √ | √ | √ | | | | | | √ | √ | √ | √ | √ | | | √ | | √ |
| WinEnableWindowUpdate | | | | | √ | | | | √ | | | | | √ | | √ | | | | | | | | | | | | | | |
| WinEndPaint | | | √ | √ | √ | √ | √ | √ | √ | √ | | √ | √ | √ | √ | | | | | | √ | √ | √ | √ | √ | | | √ | | √ |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | ANIMALS | 9 | IMAGE | 17 | REXX\REXXSAMP\CALLREXX | 25 | CLOCK |
| 2 | CLIPBRD | 10 | IPF | 18 | REXX\REXXSAMP\DEVINFO | 26 | TP |
| 3 | DIALOG | 11 | JIGSAW | 19 | REXX\REXXSAMP\RXMACDLL | 27 | VDD |
| 4 | DDLAPI | 12 | EAS | 20 | REXX\REXXSAMP\REXXUTIL | 28 | VMM |
| 5 | DRAGDROP | 13 | NPIPE | 21 | SEMAPH | 29 | WORMS |
| 6 | GRAPHIC | 14 | PRINT | 22 | SORT | 30 | WPCAR |
| 7 | HANOI | 15 | QUEUES | 23 | STYLE | | |
| 8 | HELLO | 16 | REXX\REXXSAMP\PMREXX | 24 | TEMPLATE | | |

| Function Name | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| WinGetPS | | | | √ | | √ | √ | | | √ | | √ | √ | √ | √ | √ | | | | | √ | √ | √ | | √ | | | | | |
| WinGetScreenPS | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | | | |
| WinInvalidateRect | | | | √ | | √ | √ | | √ | √ | | √ | √ | √ | √ | √ | | | | | √ | √ | | | √ | | | √ | | √ |
| WinInvalidateRegion | | | √ | | | √ | | √ | | | | | | √ | | | | | | | | | | | | | | | | |
| WinOpenWindowDC | | √ | | | | √ | | | √ | | √ | | | √ | | | | | | | | | | √ | √ | | | | | |
| WinQueryUpdateRect | | √ | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinQueryUpdateRegion | | | | | | √ | | | | | √ | | | √ | | | | | | | | | | | | | | | | |
| WinReleasePS | | | | √ | | √ | √ | | | √ | | √ | √ | √ | √ | √ | | | | | √ | √ | √ | | √ | | | | | |
| WinShowWindow | | | | | √ | | | | | | | | | √ | | √ | | | | | | | | | √ | | | | | √ |
| WinUpdateWindow | | | | | | | | | | | | | √ | √ | √ | √ | | | | | | | | | | | | | | |
| WinValidateRect | | √ | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinValidateRegion | | | | | | √ | | | | | √ | | | √ | | | | | | | | | | | | | | | | |
| **Drawing Helpers** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinDrawBitmap | | | | | | | | | √ | √ | | √ | | √ | | | | | | | | | | | | | | | | |
| WinDrawBorder | | | | | | | | | | | | | √ | | | | | | | | | | | | √ | | | | | |
| WinDrawPointer | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | √ |
| WinDrawText | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| WinFillRect | | √ | √ | √ | | √ | √ | | √ | | | √ | | √ | | | | | | | √ | | √ | √ | √ | | | √ | | √ |
| WinInvertRect | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| WinQueryPresParam | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | | | |
| WinRemovePresParam | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | | | |
| WinScrollWindow | | | | | | | | | √ | | √ | | √ | | √ | | | | | | | | | | | | | | | |
| WinSetPresParam | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | | | |
| **Error Processing** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinFreeErrorInfo | | √ | | | | √ | | √ | | | √ | | | | | | | | | | | | | | | | | | | |
| WinGetErrorInfo | | √ | | | | √ | | √ | | | √ | | | | | | | | | | | | | | | | | | | |
| WinGetLastError | | | | | | √ | | | | | √ | | | √ | | | | | | | | | | | | | | | | |
| **Help Manager** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinAssociateHelpInstance | | | | √ | √ | √ | √ | | √ | | √ | √ | √ | √ | √ | √ | | | | | √ | √ | √ | √ | √ | | | √ | | |
| WinCreateHelpInstance | | | | √ | √ | √ | √ | | √ | | √ | √ | √ | √ | √ | √ | | | | | √ | √ | √ | √ | √ | | | √ | | |
| WinDestroyHelpInstance | | | | √ | √ | √ | √ | | √ | | √ | √ | √ | √ | √ | √ | | | | | √ | √ | √ | √ | √ | | | √ | | |
| **Initialization and Termination** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinInitialize | | √ | √ | √ | √ | √ | √ | √ | √ | | √ | √ | √ | √ | √ | √ | | | | √ | √ | √ | √ | √ | √ | | | √ | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | ANIMALS | 9 | IMAGE | 17 | REXX\REXXSAMP\CALLREXX | 25 | CLOCK |
| 2 | CLIPBRD | 10 | IPF | 18 | REXX\REXXSAMP\DEVINFO | 26 | TP |
| 3 | DIALOG | 11 | JIGSAW | 19 | REXX\REXXSAMP\RXMACDLL | 27 | VDD |
| 4 | DDLAPI | 12 | EAS | 20 | REXX\REXXSAMP\REXXUTIL | 28 | VMM |
| 5 | DRAGDROP | 13 | NPIPE | 21 | SEMAPH | 29 | WORMS |
| 6 | GRAPHIC | 14 | PRINT | 22 | SORT | 30 | WPCAR |
| 7 | HANOI | 15 | QUEUES | 23 | STYLE | | |
| 8 | HELLO | 16 | REXX\REXXSAMP\PMREXX | 24 | TEMPLATE | | |

| Function Name | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| WinQueryAnchorBlock | | | | | | | | | | √ | | | | | | √ | | | | √ | | | | | | | | | | √ |
| **Keyboard** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinGetKeyState | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| WinQueryFocus | | | | | | | | | | | √ | | | | | | | | | | | | √ | | √ | | | | | |
| WinSetFocus | | √ | | | √ | | | | | | √ | √ | | | | √ | | | | | | | √ | | √ | | | | | √ |
| **List Box** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinDeleteLboxItem | | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinInsertLboxItem | | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinQueryLboxCount | | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinQueryLboxItemText | | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Menus** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinEnableMenuItem | | | | | | | | | | | | | √ | | | | | | | | | | | | | | | | | |
| WinLoadMenu | | | | | | | | | | | | | | | | | | | | | | | √ | | √ | | | | | |
| WinPopupMenu | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | | | |
| **Message Management** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinCreateMsgQueue | | √ | √ | √ | √ | √ | √ | √ | √ | | √ | √ | √ | √ | √ | √ | | | | | √ | √ | √ | √ | √ | | | √ | | |
| WinDestroyMsgQueue | | √ | √ | √ | √ | √ | √ | √ | √ | | √ | √ | √ | √ | √ | √ | | | | | √ | √ | √ | √ | √ | | | √ | | |
| WinDispatchMsg | | √ | √ | √ | √ | √ | √ | √ | √ | | √ | √ | √ | √ | √ | √ | | | | | √ | √ | √ | √ | √ | | | √ | | |
| WinGetMsg | | √ | √ | √ | √ | √ | √ | √ | √ | | √ | √ | √ | √ | √ | √ | | | | | √ | √ | √ | √ | √ | | | √ | | |
| WinLoadMessage | | | | √ | √ | | √ | | √ | | | √ | √ | | √ | | | | | | √ | √ | √ | √ | √ | | | √ | | |
| WinPeekMsg | | | | | | √ | | | | | √ | | | √ | | | | | | | | | | | | | | | | |
| WinPostMsg | | √ | √ | | √ | √ | √ | √ | √ | | √ | √ | √ | √ | √ | √ | | | | | √ | √ | √ | √ | √ | | | | | √ |
| WinPostQueueMsg | | | | | | √ | | | | | √ | | | √ | | | | | | | | | | | | | | | | |
| WinSendDlgItemMsg | | | √ | √ | √ | √ | √ | | | | | √ | | √ | | | | | | | | √ | √ | | √ | | | √ | | √ |
| WinSendMsg | | √ | √ | √ | √ | √ | √ | | √ | √ | √ | √ | √ | √ | √ | √ | | | | | √ | √ | √ | √ | √ | | | √ | | √ |
| **Mouse Capture** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinQueryCapture | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| WinSetCapture | | | | | | | | | | | √ | | | | | | | | | | | | | | | | | | | |
| **Mouse Tracking** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinShowTrackRect | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinTrackRect | | √ | | | | | | | √ | | | | | | | | | | | | | | | | √ | | | | | |
| **Pointer** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinDestroyPointer | | | | | √ | | | | | | | | | | | | | | | | | | | | | | | | | √ |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | ANIMALS | 9 | IMAGE | 17 | REXX\REXXSAMP\CALLREXX | 25 | CLOCK |
| 2 | CLIPBRD | 10 | IPF | 18 | REXX\REXXSAMP\DEVINFO | 26 | TP |
| 3 | DIALOG | 11 | JIGSAW | 19 | REXX\REXXSAMP\RXMACDLL | 27 | VDD |
| 4 | DDLAPI | 12 | EAS | 20 | REXX\REXXSAMP\REXXUTIL | 28 | VMM |
| 5 | DRAGDROP | 13 | NPIPE | 21 | SEMAPH | 29 | WORMS |
| 6 | GRAPHIC | 14 | PRINT | 22 | SORT | 30 | WPCAR |
| 7 | HANOI | 15 | QUEUES | 23 | STYLE | | |
| 8 | HELLO | 16 | REXX\REXXSAMP\PMREXX | 24 | TEMPLATE | | |

| Function Name | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| WinLoadPointer | | | | | √ | | | | | | | | | | | | | | | | | | √ | | | | | | | √ |
| WinQueryPointer | | | | | | √ | | | | | | | | √ | | | | | | | | | √ | | | | | | | |
| WinQueryPointerPos | | | | | | √ | | | | | | | | √ | | | | | | | | | √ | | | | | | | |
| WinQuerySysPointer | | √ | | | √ | √ | | | √ | | | | | √ | | √ | | | | | | | | | | | | | | |
| WinSetPointer | | √ | | | √ | √ | | | √ | | | | | √ | | √ | | | | | | | | | | | | | | |
| WinShowPointer | | | | | | | | | | | √ | | | | | | | | | | | | | | | | | | | |
| **Rectangles** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinCopyRect | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| WinIntersectRect | | | | | | | | | | | √ | | | | | | | | | | | | | | | | | | | |
| WinIsRectEmpty | | √ | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinOffsetRect | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| WinPtInRect | | | | | | √ | | | | | √ | | | √ | | | | | | | | | | | √ | | | | | |
| WinSetRect | | √ | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| WinUnionRect | | | | | | | | | | | √ | | | | | | | | | | | | | | | | | | | |
| **Standard Window** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinCalcFrameRect | | | | | | | | | √ | | | | | | | | | | | | | | | | | | | | | |
| WinCreateStdWindow | | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | | | | | √ | √ | √ | √ | √ | | | √ | | √ |
| **String/Character and Code Pages** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinLoadString | | √ | √ | √ | √ | √ | √ | √ | √ | | √ | √ | √ | √ | √ | √ | | | | | √ | √ | √ | √ | √ | | | √ | | √ |
| WinSetCp | | | | | | | | | | | | | | | | √ | | | | | | | | | | | | | | |
| **Task List** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinAddSwitchEntry | | | | | | √ | | | | | √ | | | √ | | | | | | | | | | | √ | | | | | |
| WinChangeSwitchEntry | | | | | | | | | | | √ | | | | | √ | | | | | | | | | | | | | | |
| WinQuerySwitchEntry | | | | | | | | | | | | | | | | √ | | | | | | | | | | | | | | |
| WinQuerySwitchHandle | | | | | | | | | | | | | | | | √ | | | | | | | | | | | | | | |
| WinQueryTaskSizePos | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | √ |
| WinQueryTaskTitle | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | | | |
| WinRemoveSwitchEntry | | | | | | | | | | | | | | | | √ | | | | | | | | | √ | | | | | |
| **System Values** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinQuerySysValue | | √ | | | | √ | | | √ | | √ | | | √ | | √ | | | | | | | | | √ | | | | | |
| **Timers** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinStartTimer | | | | | √ | √ | | | | √ | | | | √ | | √ | | | | | | | | | √ | | | | | √ |
| WinStopTimer | | | | | √ | √ | | | | √ | | | | √ | | √ | | | | | | | | | | | | | | √ |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | ANIMALS | 9 | IMAGE | 17 | REXX\REXXSAMP\CALLREXX | 25 | CLOCK |
| 2 | CLIPBRD | 10 | IPF | 18 | REXX\REXXSAMP\DEVINFO | 26 | TP |
| 3 | DIALOG | 11 | JIGSAW | 19 | REXX\REXXSAMP\RXMACDLL | 27 | VDD |
| 4 | DDLAPI | 12 | EAS | 20 | REXX\REXXSAMP\REXXUTIL | 28 | VMM |
| 5 | DRAGDROP | 13 | NPIPE | 21 | SEMAPH | 29 | WORMS |
| 6 | GRAPHIC | 14 | PRINT | 22 | SORT | 30 | WPCAR |
| 7 | HANOI | 15 | QUEUES | 23 | STYLE | | |
| 8 | HELLO | 16 | REXX\REXXSAMP\PMREXX | 24 | TEMPLATE | | |

| Function Name | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Window Management** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Activation, Size and Position | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinQueryWindowPos | | | | | | √ | | | | | | √ | | √ | | √ | | | | | √ | | | | √ | | | | | |
| WinSetActiveWindow | | | | | | | | | | | | | √ | | | | | | | | | | | | | | | | | |
| WinSetMultWindowPos | | | | | | | | | | | | | | | | | | | | | | | | | √ | | | | | |
| WinSetWindowPos | | | | | √ | √ | | √ | √ | | √ | | √ | √ | | √ | | | | | | | √ | | √ | | | | | √ |
| Creation and Class Information | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinCreateWindow | | | | √ | √ | √ | √ | | | | | √ | √ | √ | √ | √ | | | | | | | √ | √ | | | | | | |
| WinDefWindowProc | | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | | | | | √ | √ | √ | √ | √ | | | √ | | √ |
| WinDestroyWindow | | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | | | | | √ | √ | √ | √ | √ | | | √ | | √ |
| WinRegisterClass | | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ | | | | | √ | √ | √ | √ | √ | | | √ | | √ |
| WinSubclassWindow | | | | | √ | | | | √ | | | | | | | √ | | | | | | | | | √ | | | | | |
| General Window Information | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinEnableWindow | | | | | √ | | | | √ | | √ | √ | √ | √ | √ | √ | | | | | | | | | | | | | | |
| WinIsWindow | | | | | | | √ | | √ | | | | √ | | √ | | | | | | | √ | √ | √ | | | | | | |
| WinQueryWindowDC | | | | | | √ | | | | | | | | √ | | | | | | | | | | | | | | | | |
| WinQueryWindowProcess | | | | | | √ | | | | | √ | | | √ | | √ | | | | | | | | | √ | | | | | |
| WinQueryWindowRect | | √ | | √ | √ | √ | √ | | √ | | √ | √ | √ | √ | √ | | | | | | | √ | √ | | √ | | | √ | | √ |
| WinWindowFromID | | √ | √ | √ | √ | √ | √ | | √ | | | √ | | | | √ | | | | | | | | | | | | | | |
| Window Hierarchies | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinIsChild | | | | | | | | | √ | | | | | | | | | | | | | | | | | | | | | |
| WinQueryWindow | | | | | | | √ | | | √ | | √ | √ | | √ | √ | | | | | | √ | | | √ | | | √ | | √ |
| WinSetParent | | | | | | | | | √ | | | | | | | | | | | | | | | | √ | | | | | |
| **Window Text** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinQueryDlgItemShort | | | | | | | √ | | | | | | | | | | | | | | √ | √ | | | | | | | | |
| WinQueryDlgItemText | | | | √ | | | | | | | | √ | | | | | | | | | | | | | √ | | | √ | | |
| WinQueryWindowText | | √ | √ | | √ | | | | | | | | √ | | √ | √ | | | | | | | √ | | √ | | | | | |
| WinSetDlgItemShort | | | | | | | √ | | | | | | | √ | | | | | | | | √ | | | | | | | | |
| WinSetDlgItemText | | | | √ | √ | | | | | | √ | √ | | | | √ | | | | | √ | | √ | | √ | | | √ | | √ |
| WinSetWindowText | | √ | √ | | √ | √ | | √ | √ | | √ | | √ | √ | √ | √ | | | | | √ | | √ | √ | √ | | | | | |
| **Window Words** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WinQueryWindowPtr | | | | | | √ | | | | | | | | √ | | √ | | | | | | | | | | | | | | |
| WinQueryWindowULong | | | | | | | | | | √ | | | | | | | | | | | | | √ | | | | | | | √ |

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | ANIMALS | 9 | IMAGE | 17 | REXX\REXXSAMP\CALLREXX | 25 | CLOCK |
| 2 | CLIPBRD | 10 | IPF | 18 | REXX\REXXSAMP\DEVINFO | 26 | TP |
| 3 | DIALOG | 11 | JIGSAW | 19 | REXX\REXXSAMP\RXMACDLL | 27 | VDD |
| 4 | DDLAPI | 12 | EAS | 20 | REXX\REXXSAMP\REXXUTIL | 28 | VMM |
| 5 | DRAGDROP | 13 | NPIPE | 21 | SEMAPH | 29 | WORMS |
| 6 | GRAPHIC | 14 | PRINT | 22 | SORT | 30 | WPCAR |
| 7 | HANOI | 15 | QUEUES | 23 | STYLE | | |
| 8 | HELLO | 16 | REXX\REXXSAMP\PMREXX | 24 | TEMPLATE | | |

| Function Name | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| WinQueryWindowUShort | | | | | | | | | | | √ | | √ | √ | | √ | | | | | | | √ | | | | | | | |
| WinSetWindowPtr | | | | | | √ | | | | | | | | √ | | √ | | | | | | | | | | | | | | |
| WinSetWindowULong | | | | | | | | | | √ | | | | | | | | | | | | | | | | | | | | √ |
| WinSetWindowUShort | | | | | | | | | | | | | √ | | | √ | | | | | | | | | | | | | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | ANIMALS | 9 | IMAGE | 17 | REXX\REXXSAMP\CALLREXX | 25 | CLOCK |
| 2 | CLIPBRD | 10 | IPF | 18 | REXX\REXXSAMP\DEVINFO | 26 | TP |
| 3 | DIALOG | 11 | JIGSAW | 19 | REXX\REXXSAMP\RXMACDLL | 27 | VDD |
| 4 | DDLAPI | 12 | EAS | 20 | REXX\REXXSAMP\REXXUTIL | 28 | VMM |
| 5 | DRAGDROP | 13 | NPIPE | 21 | SEMAPH | 29 | WORMS |
| 6 | GRAPHIC | 14 | PRINT | 22 | SORT | 30 | WPCAR |
| 7 | HANOI | 15 | QUEUES | 23 | STYLE | | |
| 8 | HELLO | 16 | REXX\REXXSAMP\PMREXX | 24 | TEMPLATE | | |

# Index


## A

accelerator table 1-21
access instance data, SOM macros 7-17
accessing data, SOM 7-20
active window 1-19
address format
  segment:offset 6-1
  selector:offset 6-1
anchor block 3-4
anchor-block handle 3-4
application compatibility 6-5
application queue 1-20
application resources 1-21
application restrictions 6-9
application-controlled viewports 4-40
application-generated data exchange 1-23
application-oriented user environments 8-1
arc 1-25
area 1-25
asynchronous communications BIOS functions 6-15
available file handles 4-34
available memory 4-11

## B

BadDynLink 4-29
base extender functions 6-12
bi-modal device driver code 6-3
bimodal device drivers 1-2
BIOS 6-15
bit map 1-21, 1-26, 4-51
BKM_CALCPAGERECT 4-45
BKM_DELETEPAGE 4-45
BKM_INSERTPAGE 4-45
BKM_INVALIDATETABS 4-45
BKM_QUERYPAGECOUNT 4-45
BKM_QUERYPAGEID 4-45
BKM_QUERYPAGEULONG 4-45
BKM_QUERYPAGEWINDOWHWND 4-45
BKM_QUERYTABBITMAP 4-45
BKM_QUERYTABTEXT 4-45
BKM_SETDIMENSIONS 4-45
BKM_SETPAGEULONG 4-45
BKM_SETPAGEWINDOWHWND 4-45
BKM_SETSTATUSLINETEXT 4-45
BKM_SETTABBITMAP 4-45
BKM_SETTABTEXT 4-45
BKM_TURNTOPAGE 4-45
BKN_NEWPAGESIZE 4-45
BKN_PAGESELECTED 4-45
BKS_BACKPAGESBL 4-44
BKS_BACKPAGESBR 4-44
BKS_BACKPAGESTL 4-44
BKS_BACKPAGESTR 4-44
BKS_MAJORTABBOTTOM 4-45
BKS_MAJORTABLEFT 4-45
BKS_MAJORTABRIGHT 4-45
BKS_MAJORTABTOP 4-45
BKS_POLYGONTABS 4-45
BKS_ROUNDEDTABS 4-45
BKS_SQUARETABS 4-45
BKS_STATUSTEXTCENTER 4-45
BKS_STATUSTEXTLEFT 4-45
BKS_STATUSTEXTRIGHT 4-45
block device 1-12
building a class library, .DEF 7-12
bus interface unit 2-2
byte-addressable segment 2-14

## C

C program template for class implementation, .C 7-12
caching mechanisms
  Translation Lookaside Buffer (TLB) 2-7
calling 16-bit code, linkage conventions 3-19
character device 1-9, 1-12
Character Output, SOM 7-19
CheckMsgFilterHook 4-51
child class 7-4
child process 1-6
class 7-4
class definition file
  class section 7-7
  Comments 7-10
  data section 7-7
  include section 7-7
  metaclass section 7-7
  methods section 7-7
  parent class section 7-7
  passthru section 7-7
  release order section 7-7, 7-8
class hierarchy, Workplace objects 8-8
class implementation
  .C, SOM C-language binding file 7-12
  .IH, implementation header file 7-12
  .SC, public class implementation file 7-12
class implementation file, .IH 7-22
class libraries, SOM 7-32
Class List Utility, Workplace 8-45
class method 7-5
class object 7-4, 7-5
class section, class definition file 7-7
class-specific SOM Macros 7-16
classes of user interface objects
  container object 8-3
  data object 8-3

classes of user interface objects *(continued)*
device object 8-3
CLASSFIELDINFO 8-24
client applications, SOM 7-6
client process 1-10
client program, SOM 7-12, 7-23, 7-26
clipboard 1-22
clipped window 1-14
clipping 1-27
Clock object 8-7
CLSSTYLE_NEVERCOPY 8-36
CLSSTYLE_NEVERDELETE 8-36
CLSSTYLE_NEVERDRAG 8-36
CLSSTYLE_NEVERMOVE 8-36
CLSSTYLE_NEVERPRINT 8-36
CLSSTYLE_NEVERRENAME 8-36
CLSSTYLE_NEVERSHADOW 8-36
CLSSTYLE_NEVERTEMPLATE 8-36
CLSSTYLE_NEVERVISIBLE 8-36
CMOS/Real Time Clock 6-15
CM_ALLOCDETAILFIELDINFO 4-47
CM_ALLOCRECORD 4-47
CM_ARRANGE 4-47
CM_CLOSEEDIT 4-47
CM_ERASERECORD 4-47
CM_FILTER 4-47
CM_FREEDETAILFIELDINFO 4-47
CM_FREERECORD 4-47
CM_HORZSCROLLSPLITWINDOW 4-47
CM_INSERTDETAILFIELDINFO 4-47
CM_INSERTRECORD 4-47
CM_INVALIDATEDETAILFIELDINFO 4-47
CM_INVALIDATERECORD 4-47
CM_OPENEDIT 4-47
CM_PAINTBACKGROUND 4-47
CM_QUERYCNRINFO 4-47
CM_QUERYDETAILFIELDINFO 4-47
CM_QUERYDRAGIMAGE 4-47
CM_QUERYRECORD 4-47
CM_QUERYRECORDEMPHASIS 4-47
CM_QUERYRECORDFROMRECT 4-47
CM_QUERYRECORDRECT 4-47
CM_QUERYVIEWPORTRECT 4-47
CM_REMOVEDETAILFIELDINFO 4-47
CM_REMOVERECORD 4-47
CM_SCROLLWINDOW 4-47
CM_SEARCHSTRING 4-47
CM_SETCNRINFO 4-47
CM_SETRECORDEMPHASIS 4-47
CM_SORTRECORD 4-47
CN_DRAGAFTER 4-46
CN_DRAGLEAVE 4-46
CN_DRAGOVER 4-46
CN_DROP 4-46
CN_DROPHELP 4-46
CN_EMPHASIS 4-46
CN_ENDEDIT 4-46
CN_ENTER 4-46
CN_INITDRAG 4-46
CN_KILLFOCUS 4-46
CN_QUERYDELTA 4-46
CN_REALLOCPSZ 4-46
CN_SCROLL 4-46
CN_SETFOCUS 4-46
CODE attributes
LOADONCALL 5-6
PRELOAD 5-6
code page management 4-34
code prefetch unit 2-2
code segment 2-6
allowable types 2-8
defining 5-13
CODE statement, module-definition file 5-6
colliding method names 7-17
color index 1-26
Color Palette object 8-7
COM 6-15
comments, class definition file 7-10
compaction 4-1
compatibility region 3-16
compatibility, SOM 7-8
compiler support 3-11
compiling a program
using the /Gt+ option 3-10
using the /Sp option 3-23
using the /Sp2 option 3-23
composed view of objects 8-2
concurrent DOS sessions 6-5
limit 6-5
CONCURRENTVIEW 8-34
conditional cascade menus 8-12
CONFIG.SYS 6-7, 6-9
constructor, class 7-5
container control window 8-23
container object 8-3
container window control 4-45
messages
CM_ALLOCDETAILFIELDINFO 4-47
CM_ALLOCRECORD 4-47
CM_ARRANGE 4-47
CM_CLOSEEDIT 4-47
CM_ERASERECORD 4-47
CM_FILTER 4-47
CM_FREEDETAILFIELDINFO 4-47
CM_FREERECORD 4-47
CM_HORZSCROLLSPLITWINDOW 4-47
CM_INSERTDETAILFIELDINFO 4-47
CM_INSERTRECORD 4-47
CM_INVALIDATEDETAILFIELDINFO 4-47
CM_INVALIDATERECORD 4-47
CM_OPENEDIT 4-47
CM_PAINTBACKGROUND 4-47
CM_QUERYCNRINFO 4-47
CM_QUERYDETAILFIELDINFO 4-47
CM_QUERYDRAGIMAGE 4-47
CM_QUERYRECORD 4-47

container window control *(continued)*
messages *(continued)*
CM_QUERYRECORDEMPHASIS 4-47
CM_QUERYRECORDFROMRECT 4-47
CM_QUERYRECORDRECT 4-47
CM_QUERYVIEWPORTRECT 4-47
CM_REMOVEDETAILFIELDINFO 4-47
CM_REMOVERECORD 4-47
CM_SCROLLWINDOW 4-47
CM_SEARCHSTRING 4-47
CM_SETCNRINFO 4-47
CM_SETRECORDEMPHASIS 4-47
CM_SORTRECORD 4-47
CN_DRAGAFTER 4-46
CN_DRAGLEAVE 4-46
CN_DRAGOVER 4-46
CN_DROP 4-46
CN_DROPHELP 4-46
CN_EMPHASIS 4-46
CN_ENDEDIT 4-46
CN_ENTER 4-46
CN_INITDRAG 4-46
CN_KILLFOCUS 4-46
CN_QUERYDELTA 4-46
CN_REALLOCPSZ 4-46
CN_SCROLL 4-46
CN_SETFOCUS 4-46
styles
CS_AUTOPOSITION 4-46
CS_AUTOSELECTION 4-46
CS_EXTENDSEL 4-46
CS_MULTIPLESEL 4-46
CS_READONLY 4-46
CS_SINGLESEL 4-46
CS_VERIFYPOINTERS 4-46
contents view of objects
details view of an object 8-2
icons view 8-2
control program functions 1-29
control windows 1-16
coordinate space 1-27
coprocessing 2-13
Country object 8-7
Create another action 8-22
critical sections 4-14
CSC, extension of class definition file 7-6
CS_AUTOPOSITION 4-46
CS_AUTOSELECTION 4-46
CS_EXTENDSEL 4-46
CS_MULTIPLESEL 4-46
CS_READONLY 4-46
CS_SINGLESEL 4-46
CS_VERIFYPOINTERS 4-46
CTRL+BREAK 4-35, 4-36
CTRL+C 4-35, 4-36
CTXT_CLOSE 8-13
CTXT_COPY 8-13
CTXT_CREATEANOTHER 8-13
CTXT_DELETE 8-13
CTXT_HELP 8-13
CTXT_MOVE 8-13
CTXT_OPEN 8-13
CTXT_PRINT 8-13
CTXT_SETTINGS 8-13
CTXT_SHADOW 8-13
CTXT_WINDOW 8-13
CTXT_, standard pop-up menu items 8-13
cursor 1-19

# D

data object 8-3
data section, class definition file 7-7
data segment 2-6
allowable types 2-8
defining 5-13
DATA statement 5-8
definition 5-8
options
LOADONCALL 5-8
MULTIPLE 5-8
PRELOAD 5-8
READONLY 5-8
READWRITE 5-8
SINGLE 5-8
DC_SEM_SHARED 4-18
DdfBeginList 4-41
DdfBitmap 4-41
DdfEndList 4-41
DdfHyperText 4-41
DdfInform 4-41
DdfInitialize 4-41
DdfListItem 4-41
DdfMetafile 4-41
DdfPara 4-41
DdfSetColor 4-41
DdfSetFont 4-41
DdfSetFontStyle 4-41
DdfSetFormat 4-41
DdfSetTextAlign 4-41
DdfText 4-41
debugging programs 4-16
debugging, SOM 7-18
dedicated paging unit 2-7
Default Class Styles for Objects
CLSSTYLE_NEVERCOPY 8-36
CLSSTYLE_NEVERDELETE 8-36
CLSSTYLE_NEVERDRAG 8-36
CLSSTYLE_NEVERMOVE 8-36
CLSSTYLE_NEVERPRINT 8-36
CLSSTYLE_NEVERRENAME 8-36
CLSSTYLE_NEVERSHADOW 8-36
CLSSTYLE_NEVERTEMPLATE 8-36
CLSSTYLE_NEVERVISIBLE 8-36

default dialog procedure 1-18
descriptor table 2-5
descriptor table entry 2-5
  granularity bit 2-5
designing an object-oriented user interface 8-4
designing workplace classes 8-9
Desktop 8-6
desktop window 1-14
DestroyWindowHook 4-51
details view 8-23
details view of an object 8-2
device context 1-4, 1-24, 1-28
  definition of 1-25
device driver code
  bi-modal 6-3
device drivers 1-12
device emulation 6-4
device I/O 4-30
device object 8-3
DEVICE statement 6-7, 6-9
device support 1-12
device virtualization 6-4
device-helper services 1-12
device-independent graphics 1-4
devices 1-12
  block 1-12
  character 1-12
dialog
  box editor 1-22
  includes 1-21
  procedure 1-18
  template 1-17
  templates 1-21
  window 1-17
dialog box 1-18
direct manipulation 1-23
direct manipulation functions 1-29
Direct Manipulation Methods 8-9
direct manipulation, drag/drop 8-4, 8-28
direct memory access (DMA) 1-12
direct printing 1-29
directories, searching 4-32
directory 1-8
discarding memory objects 4-11
disk/diskette 6-15
dispatching priority 1-6
DLL 5-2
  defining code attributes 5-6
  defining data attributes 5-8
  global data 5-7
  initialization 5-9
    INITGLOBAL option 5-13
    INITINSTANCE option 5-13
    LIBRARY statement 5-9
  instance data 5-7
  shared data 5-7
  termination 5-9
DLL Management, SOM 7-19
DMA controller 6-15
DM&Us.ENDCONVERSATION 8-29
DM_ 8-28
DM_DRAGOVER 8-29
DM_DROP 8-29
DM_PRINTOBJECT 8-28
DM_RENDER 8-29
DM_RENDERCOMPLETE 8-29
DOS application capabilities 6-5
DOS applications
  displaying under PM 6-1
  multiple sessions 6-1
  running under OS/2 6-1
  running under PM 6-6
DOS compatibility environment 1-11, 6-1
DOS compatibility mode 6-1
DOS Emulation 6-2
DOS extender services 6-11
DOS interrupt INT 67h 6-7
DOS Protect Mode Interface (DPMI) 6-10
DOS Session 6-1
  capabilities 6-5
  definition 6-2
  DOS Settings 6-5
  I/O privilege map 6-2
  software interrupt reflection 6-2
  specifying limits 6-7
DOS Session Manager 6-2
DOS Settings 6-5
  categories
    DOS environment 6-5
    file operations 6-5
    hardware environment 6-5
    memory extenders 6-5
    video 6-5
    windowing 6-5
DosAcknowledgeSignalException 4-36
DosAddMuxWaitSem 4-19, 4-25
DosAllocHuge 4-2
DosAllocMem 4-2, 4-9
DosAllocSeg 4-2
DosAllocSharedMem 4-2, 4-6
DosAllocShrSeg 4-2, 4-6
DosAsyncTimer 4-28
DosBeep 4-30
DosBufReset 4-31
DosCallBack 4-12, 4-17
DosCallNmPipe 4-27
DosCallNPipe 4-27
DOSCALL1.DLL 5-3
DosCancelLockRequest 4-32
DosCaseMap 4-34
DosChDir 4-31
DosChgFilePtr 4-31
DosCLIAccess 4-30
DosClose 4-31

DosCloseEventSem 4-19, 4-20
DosCloseMutexSem 4-19, 4-21
DosCloseMuxWaitSem 4-19, 4-25
DosCloseQueue 4-28
DosCloseSem 4-19
DosCloseVDD 4-37
DosConnectNmPipe 4-27
DosConnectNPipe 4-27
DosCopy 4-31
DosCreateCSAlias 4-2, 4-9
DosCreateDir 4-31
DosCreateEventSem 4-19, 4-20
DosCreateMutexSem 4-19, 4-21
DosCreateMuxWaitSem 4-19, 4-25
DosCreateNmPipe 4-27
DosCreateNPipe 4-27
DosCreatePipe 4-26
DosCreateQueue 4-28
DosCreateSem 4-19
DosCreateThread 4-12
DosCreateThread, new features 4-12
DosCWait 4-12, 4-14
DosDebug 4-12, 4-16
DosDelete 4-31
DosDeleteDir 4-31
DosDeleteMuxWaitSem 4-19, 4-25
DosDevConfig 4-30
DosDevIOCtl 4-30
DosDevIOCtl2 4-30
DosDisConnectNmPipe 4-27
DosDisConnectNPipe 4-27
DosDupHandle 4-31
DosEditName 4-31
DosEnterCritSec 4-12, 4-14
DosEnterMustComplete 4-36
DosEnumAttribute 4-31
DosErrClass 4-35
DosError 4-35
DosExecPgm 4-12, 4-16
DosExit 4-12, 4-13
DosExitCritSec 4-12, 4-14
DosExitList 4-12
DosExitMustComplete 4-36
DosFileIO 4-31
DosFileLocks 4-31
DosFindClose 4-31, 4-32
DosFindFirst 4-31, 4-32
DosFindNext 4-31, 4-32
DosFindNotifyClose 4-31
DosFindNotifyFirst 4-31
DosFindNotifyNext 4-31
DosFlatToSel 3-14, 3-17
DosFreeMem 4-2, 4-4
DosFreeModule 4-29, 5-6
DosFreeResource 4-29
DosFreeSeg 4-2
DosFSAttach 4-31
DosFSCtl 4-31
DosFSRamSemClear 4-19
DosFSRamSemRequest 4-19
DosGetCollate 4-34
DosGetCp 4-34
DosGetCtryInfo 4-34
DosGetDateTime 4-28
DosGetDBCSEv 4-34
DosGetEnv 4-12, 4-14, 4-29
DosGetHugeShift 4-2
DosGetInfoBlocks 4-12, 4-14
DosGetInfoSeg 4-12, 4-14
DosGetMachineMode 4-29
DosGetMessage 4-34
DosGetModeName 4-29
DosGetModHandle 4-29
DosGetNamedSharedMem 4-2, 4-6
DosGetPID 4-12, 4-14
DosGetPPID 4-12, 4-14
DosGetProcAddr 4-29
DosGetPrty 4-12, 4-14
DosGetResource 4-29
DosGetResource2 4-29
DosGetSeg 4-2, 4-8
DosGetSharedMem 4-2, 4-8
DosGetShrSeg 4-2, 4-6
DosGetVersion 4-29
DosGiveSeg 4-2, 4-8
DosGiveSharedMem 4-2, 4-8
DosHoldSignal 4-35
DosInsertMessage 4-34
DosInsMessage 4-34
DosKillProcess 4-12, 4-14
DosLoadModule 4-29
DosLoadModule, runtime dynamic linking 5-6
DosLockSeg 4-2, 4-11
DosMakePipe 4-26
DosMapCase 4-34
DosMemAvail 4-2, 4-11
DosMkDir 4-31
DosMove 4-31
DosMuxSemWait 4-19
DosNewSize 4-31
DosOpen 4-31
DosOpenEventSem 4-19, 4-20
DosOpenMutexSem 4-19, 4-21
DosOpenMuxWaitSem 4-19, 4-25
DosOpenQueue 4-28
DosOpenSem 4-19
DosOpenVDD 4-37
DosPeekNmPipe 4-27
DosPeekNPipe 4-27
DosPeekQueue 4-28
DosPhysicalDisk 4-30
DosPortAccess 4-30
DosPostEventSem 4-19, 4-20
DosPrintDestAdd 4-39

DosPrintDestControl 4-39
DosPrintDestDel 4-39
DosPrintDestEnum 4-39
DosPrintDestGetInfo 4-39
DosPrintDestSetInfo 4-39
DosPrintDriverEnum 4-39
DosPrintJobContinue 4-39
DosPrintJobDel 4-39
DosPrintJobEnum 4-39
DosPrintJobGetId 4-39
DosPrintJobGetInfo 4-39
DosPrintJobPause 4-39
DosPrintJobSetInfo 4-39
DosPrintPortEnum 4-39
DosPrintQAdd 4-39
DosPrintQContinue 4-39
DosPrintQDel 4-39
DosPrintQEnum 4-39
DosPrintQGetInfo 4-39
DosPrintQPause 4-39
DosPrintQProcessorEnum 4-39
DosPrintQPurge 4-39
DosPrintQSetInfo 4-39
DosPTrace 4-12, 4-16
DosPurgeQueue 4-28
DosPutMessage 4-34
DosQAppType 4-29
DosQCurDir 4-31
DosQCurDisk 4-31
DosQFHandState 4-31
DosQFileInfo 4-31
DosQFileMode 4-33
DosQFSAttach 4-31
DosQFSInfo 4-31
DosQHandType 4-31
DosQNmPHandState 4-27
DosQPathInfo 4-31
DosQSysInfo 4-12, 4-31, 4-33
DosQueryAppType 4-29
DosQueryCollate 4-34
DosQueryCp 4-34
DosQueryCtryInfo 4-34
DosQueryCurrentDir 4-31
DosQueryCurrentDisk 4-31
DosQueryDBCSEnv 4-34
DosQueryEventSem 4-19, 4-20
DosQueryFHState 4-31
DosQueryFileInfo 4-31, 4-33
DosQueryFileMode 4-31
DosQueryFSAttach 4-31
DosQueryFSInfo 4-31
DosQueryHType 4-31
DosQueryMem 4-2, 4-11
DosQueryMessageCp 4-34
DosQueryModuleHandle 4-29, 5-6
DosQueryModuleName 4-29, 5-6
DosQueryMutexSem 4-19, 4-21
DosQueryMuxWaitSem 4-19, 4-25
DosQueryNmPipeInfo 4-27
DosQueryNmPipeSemState 4-27
DosQueryNPHState 4-27
DosQueryNPipeInfo 4-27
DosQueryNPipeSemState 4-27
DosQueryPathInfo 4-31
DosQueryProcAddr 4-29, 5-6
DosQueryProcType 4-29
DosQueryQueue 4-28
DosQueryResourceSize 4-29
DosQuerySysInfo 4-14, 4-31, 4-33
DosQueryVerify 4-31
DosQVerify 4-31
DosRaiseException 4-36
DosRawReadNmPipe 4-27
DosRawWriteNmPipe 4-27
DosRead 4-31
DosReadAsync 4-31, 4-33
DosReadQueue 4-28
DosReallocHuge 4-2
DosReallocSeg 4-2
DosReleaseMutexSem 4-19, 4-21
DosRequestMutexSem 4-19, 4-21
DosRequestVDD 4-37
DosResetBuffer 4-31
DosResetEventSem 4-19, 4-20
DosResumeThread 4-12, 4-13
DosRetForward 4-12
DosRmDir 4-31
DosR2StackRealloc 4-12, 4-17
DosScanEnv 4-31
DosSearchPath 4-31
DosSelectDisk 4-31
DosSelectSession 4-35
DosSelToFlat 3-17
DosSemClear 4-19
DosSemRequest 4-19
DosSemSet 4-19
DosSemSetWait 4-19
DosSemWait 4-19
DosSendSignalException 4-36
DosSetCp 4-34
DosSetCurrentDir 4-31
DosSetDateTime 4-28
DosSetDefaultDisk 4-31
DosSetExceptionHandler 4-36
DosSetFHandState 4-32
DosSetFHState 4-32
DosSetFileInfo 4-32
DosSetFileLocks 4-31
DosSetFileMode 4-32, 4-33
DosSetFilePtr 4-31
DosSetFileSize 4-31
DosSetFSInfo 4-32
DosSetMaxFH 4-32, 4-34
DosSetMem 4-2, 4-11

DosSetNmHandInfo 4-27
DosSetNmPipeSem 4-27
DosSetNPHState 4-27
DosSetNPipeSem 4-27
DosSetPathInfo 4-32
DosSetPriority 4-12
DosSetProcCp 4-34
DosSetProcessCp 4-34
DosSetPrty 4-12
DosSetRelMaxFH 4-32, 4-34
DosSetSession 4-35
DosSetSigHandler 4-35
DosSetSignalExceptionFocus 4-36
DosSetVerify 4-32
DosShutDown 4-32
DosSizeSeg 4-2
DosSleep 4-28
DosSMRegisterDD 4-35
DosStartSession 4-35, 6-5
DosStartTimer 4-28
DosStopSession 4-35
DosStopTimer 4-28
DosSubAllocMem 4-2, 4-4
DosSubFreeMem 4-2, 4-4
DosSubSetMem 4-2, 4-4
DosSubUnsetMem 4-2, 4-4
DosSuspendThread 4-12, 4-13
DosTimerAsync 4-28
DosTimerStart 4-28
DosTimerStop 4-28
DosTransactNmPipe 4-27
DosTransactNPipe 4-27
DosUnlockSeg 4-2, 4-11
DosUnsetExceptionHandler 4-36
DosUnwindException 4-36
DosWaitChild 4-12, 4-14
DosWaitEventSem 4-19, 4-20
DosWaitMuxWaitSem 4-19, 4-25
DosWaitNmPipe 4-27
DosWaitNPipe 4-27
DosWaitThread 4-12, 4-14
DosWrite 4-32
DosWriteAsync 4-32, 4-34
DosWriteQueue 4-28
DOS16 prefix for 16-bit calls 3-26
DOS/Windows applications 3-2, 3-4
DPMI
  clients 6-11
  host program 6-11
  Specification Version 0.9 6-10
  types of clients 6-11
drag/drop, direct manipulation 8-4, 8-28
drawing functions 1-27
dynamic code 4-9
dynamic data exchange 1-22
  protocol 1-23
dynamic data formatting 4-40
dynamic link library (DLL) 1-7, 1-30, 5-2
dynamic linking 1-7, 4-28, 5-2
  advantages 5-2
  load-time 5-3
  run time 5-6
dynamic-data formatting
  functions 1-29

# E

EMS
  basis for 6-6
  installation 6-7
  memory
    specifying limits 6-7
  memory areas
    managed by 6-8
  VEMM.SYS 6-7
  Virtual Expanded Memory Manager (VEMM) 6-7
encapsulation 7-3
Enhanced DOS Sessions 6-1
  components 6-2
    DOS Emulation 6-2
    DOS Session Manager 6-2
    8086 Emulation 6-2
  kernel 6-2
enhanced instruction set 1-1
Error Handling Methods 8-9
Error Handling, SOM 7-19
error management 4-35
ERROR_SEM_OWNER_DIED 4-18
event semaphores 1-10
  states 4-20
exception handling 1-11
exception management 4-36
execution unit 2-2
EXEC_ASYNCRESULT 4-14
exit lists 1-6
Expanded Memory Specification (EMS) 6-6, 6-15
EXPENTRY keyword 3-29
EXPORTS statement, module-definition file 5-10
extended attributes, Workplace
  .ASSOCTABLE 8-46
  .LONGNAME 8-46
Extended Memory Specification (XMS) 6-6, 6-8, 6-15
external functions
  EXPORTS 5-10
  IMPORTS 5-10
  references 5-10
external routines 5-1

# F

factory method, class 7-5
far16 cdecl 3-19
far16 fastcall 3-19
far16 keyword 3-29

far16 pascal 3-19
fast mode switching 6-3
fast-safe RAM semaphores 4-17
file allocation table (FAT) 1-8
file dialogs 4-42
file handles, available 4-34
file mode
  querying 4-33
  setting 4-33
file sharing 1-9
file system
  character device 1-9
  device handle 1-9
  directory 1-8
  extended attributes 1-8
  file locking 1-8
  functions 4-30
  handle 1-8
  hierarchy 1-8
  root directory 1-8
  subdirectory 1-8
files
  reading asynchronously 4-33
  writing asynchronously 4-34
filtering input 1-20
FindObject class methods
  wpclsFindObjectEnd 8-37
  wpclsFindObjectFirst 8-37
  wpclsFindObjectNext 8-37
fixed high threads 1-7
flags for pop-up menu items inherited from WPObject
  CTXT_CLOSE 8-13
  CTXT_COPY 8-13
  CTXT_CREATEANOTHER 8-13
  CTXT_DELETE 8-13
  CTXT_HELP 8-13
  CTXT_MOVE 8-13
  CTXT_OPEN 8-13
  CTXT_PRINT 8-13
  CTXT_SETTINGS 8-13
  CTXT_SHADOW 8-13
  CTXT_WINDOW 8-13
flat memory model 2-13
  address range 2-6
  architecture 1-1
  code segment 2-6
  data segment 2-6
flat selectors 3-15
folders, Workplace 8-7
Font Palette object 8-7
fonts 1-21, 1-26, 4-51
  dialog boxes 4-43
  editor 1-22
frame window 1-17
freeing memory 4-4
full-screen applications 1-2, 1-4, 3-2
full-screen session 3-2

Functions
  Bit Maps
    GpiDrawBits 4-51
  Characters
    GpiQueryCharBreakExtra 4-52
    GpiQueryCharExtra 4-52
    GpiQueryCharOutline 4-52
    GpiQueryFaceString 4-52
    GpiSetCharBreakExtra 4-52
    GpiSetCharExtra 4-52
  Checking a Process's Virtual-Memory Map
    DosQueryMem 4-11
  Code-Page Management
    DosCaseMap 4-34
    DosGetCollate 4-34
    DosGetCp 4-34
    DosGetCtryInfo 4-34
    DosGetDBCSEv 4-34
    DosMapCase 4-34
    DosQueryCollate 4-34
    DosQueryCp 4-34
    DosQueryCtryInfo 4-34
    DosQueryDBCSEnv 4-34
    DosSetCp 4-34
    DosSetProcCp 4-34
    DosSetProcessCp 4-34
  Controlling Threads
    DosResumeThread 4-13
    DosSuspendThread 4-13
  Creating Threads
    DosCreateThread 4-12
  Customizing Help
    DdfBeginList 4-41
    DdfBitmap 4-41
    DdfEndList 4-41
    DdfHyperText 4-41
    DdfInform 4-41
    DdfInitialize 4-41
    DdfListItem 4-41
    DdfMetafile 4-41
    DdfPara 4-41
    DdfSetColor 4-41
    DdfSetFont 4-41
    DdfSetFontStyle 4-41
    DdfSetFormat 4-41
    DdfSetTextAlign 4-41
    DdfText 4-41
  Debugging Programs
    DosDebug 4-16
    DosPtrace 4-16
  Desktop Background
    WinQueryDesktopBkgnd 4-51
    WinSetDesktopBkgnd 4-51
  Determining Available Memory
    DosMemAvail 4-11
  Device I/O
    DosBeep 4-30
    DosCLIAccess 4-30
    DosDevConfig 4-30

Functions *(continued)*
Device I/O *(continued)*
DosDevIOCtl 4-30
DosDevIOCtl2 4-30
DosPhysicalDisk 4-30
DosPortAccess 4-30
Dialog Boxes
WinFileDgl 4-43
WinFontDgl 4-43
Discarding Memory Objects
DosLockSeg 4-11
DosUnlockSeg 4-11
DosAllocMem 4-2
DosGetShrSeg 4-2
DosSetFHState 4-32
Dynamic Linking
BadDynLink 4-29
DosFreeModule 4-29
DosFreeResource 4-29
DosGetEnv 4-29
DosGetMachineMode 4-29
DosGetModeName 4-29
DosGetModHandle 4-29
DosGetProcAddr 4-29
DosGetResource 4-29
DosGetResource2 4-29
DosGetVersion 4-29
DosLoadModule 4-29
DosQAppType 4-29
DosQueryAppType 4-29
DosQueryModuleHandle 4-29
DosQueryModuleName 4-29
DosQueryProcAddr 4-29
DosQueryProcType 4-29
DosQueryResourceSize 4-29
Ending Other Process
DosKillProcess 4-14
Error Management
DosErrClass 4-35
DosError 4-35
Exception Management
DosAcknowledgeSignalException 4-36
DosEnterMustComplete 4-36
DosExitMustComplete 4-36
DosRaiseException 4-36
DosSendSignalException 4-36
DosSetExceptionHandler 4-36
DosSetSignalExceptionFocus 4-36
DosUnsetExceptionHandler 4-36
DosUnwindException 4-36
Exiting from Threads and Processes
DosExit 4-13
File Initialization
WinQueryProfileData 4-38
WinQueryProfileInt 4-38
WinQueryProfileSize 4-38
WinQueryProfileString 4-38
WinWriteProfileData 4-38
WinWriteProfileString 4-38

Functions *(continued)*
File Systems
DosBufReset 4-31
DosCancelLockRequest 4-32
DosChDir 4-31
DosChgFilePtr 4-31
DosClose 4-31
DosCopy 4-31
DosCreateDir 4-31
DosDelete 4-31
DosDeleteDir 4-31
DosDupHandle 4-31
DosEditName 4-31
DosEnumAttribute 4-31
DosFileIO 4-31
DosFileLocks 4-31
DosFindClose 4-31
DosFindFirst 4-31
DosFindNext 4-31
DosFindNotifyClose 4-31
DosFindNotifyFirst 4-31
DosFindNotifyNext 4-31
DosFSAttach 4-31
DosFSCtl 4-31
DosMkDir 4-31
DosMove 4-31
DosNewSize 4-31
DosOpen 4-31
DosQCurDir 4-31
DosQCurDisk 4-31
DosQFHandState 4-31
DosQFileInfo 4-31
DosQFSAttach 4-31
DosQFSInfo 4-31
DosQHandType 4-31
DosQPathInfo 4-31
DosQSysInfo 4-31
DosQueryCurrentDir 4-31
DosQueryCurrentDisk 4-31
DosQueryFHState 4-31
DosQueryFileInfo 4-31
DosQueryFileMode 4-31
DosQueryFSAttach 4-31
DosQueryFSInfo 4-31
DosQueryHType 4-31
DosQueryPathInfo 4-31
DosQuerySysInfo 4-31
DosQueryVerify 4-31
DosQVerify 4-31
DosRead 4-31
DosReadAsync 4-31
DosResetBuffer 4-31
DosRmDir 4-31
DosScanEnv 4-31
DosSearchPath 4-31
DosSelectDisk 4-31
DosSetCurrentDir 4-31
DosSetDefaultDisk 4-31
DosSetFHandState 4-32

Functions *(continued)*
File Systems *(continued)*
DosSetFileInfo 4-32
DosSetFileLocks 4-31
DosSetFileMode 4-32
DosSetFilePtr 4-31
DosSetFileSize 4-31
DosSetFSInfo 4-32
DosSetMaxFH 4-32
DosSetPathInfo 4-32
DosSetRelMaxFH 4-32
DosSetVerify 4-32
DosShutDown 4-32
DosWrite 4-32
DosWriteAsync 4-32
Fonts
GpiLoadPublicFonts 4-52
GpiQueryLogicalFont 4-52
GpiUnloadPublicFonts 4-52
Freeing Memory
DosFreeMem 4-4
Generating Dynamic Code
DosAllocMem 4-9
DosCreateCSAlias 4-9
Getting Thread and Process Information
DosGetEnv 4-14
DosGetInfoBlocks 4-14
DosGetInfoSeg 4-14
DosGetPID 4-14
DosGetPPID 4-14
DosGetPrty 4-14
DosQuerySysInfo 4-14
Handling Critical Sections
DosEnterCritSec 4-14
DosExitCritSec 4-14
Heap Management
WinAllocMem 4-37
WinAvailMem 4-37
WinCreateHeap 4-37
WinDestroyHeap 4-37
WinFreeMem 4-37
WinLockHeap 4-37
WinReallocMem 4-37
Installed Program List
PrfAddProgram 4-37
PrfChangeProgram 4-37
PrfCreateGroup 4-37
PrfDestroyGroup 4-37
PrfQueryDefinition 4-37
PrfQueryProgramCategory 4-37
PrfQueryProgramHandle 4-37
PrfQueryProgramTitles 4-37
PrfRemoveProgram 4-37
WinAddProgram 4-37
WinCreateGroup 4-37
WinInstStartApp 4-37
WinQueryDefinition 4-37
WinQueryProgramTitles 4-37
WinTerminateApp 4-37

Functions *(continued)*
Memory Allocation
DosAllocHuge 4-2
DosAllocMem 4-2
DosAllocSeg 4-2
Memory Suballocation
DosSubUnsetMem 4-4
Memory Suballoction
DosSubAllocMem 4-4
DosSubFreeMem 4-4
DosSubSetMem 4-4
Memory-Management
DosAllocHuge 4-2
DosAllocSeg 4-2
DosAllocSharedMem 4-2
DosAllocShrSeg 4-2
DosCreateCSAlias 4-2
DosFreeMem 4-2
DosGetHugeShift 4-2
DosGetNamedSharedMem 4-2
DosGetSeg 4-2
DosGetSharedMem 4-2
DosGiveSeg 4-2
DosGiveSharedMem 4-2
DosLockSeg 4-2
DosMemAvail 4-2
DosQueryMem 4-2
DosReallocHuge 4-2
DosReallocSeg 4-2
DosSetMem 4-2
DosSizeSeg 4-2
DosSubAllocMem 4-2
DosSubFreeMem 4-2
DosSubSetMem 4-2
DosSubUnsetMem 4-2
DosUnlockSeg 4-2
Memory-Management Functions
DosFreeSeg 4-2
DosSubAllocMem 4-2
DosSubFreeMem 4-2
Message Retrieval
DosGetMessage 4-34
DosInsertMessage 4-34
DosInsMessage 4-34
DosPutMessage 4-34
DosQueryMessageCp 4-34
Migration
WinQueryClassThunkProc 4-42
WinQueryWindowMode 4-42
WinQueryWindowThunkProc 4-42
WinSetClassThunkProc 4-42
WinSetWindowThunkProc 4-42
Paths and Regions
GpiPathToRegion 4-51
Polylines
GpiPolylineDisjoint 4-52
Pop-Up Menus
WinPopUpMenu 4-51

Functions *(continued)*
Querying File Mode
DosQFileMode 4-33
DosQueryFileInfo 4-33
Querying System Information
DosQSysInfo 4-33
DosQuerySysInfo 4-33
Reading Asynchronously
DosReadAsync 4-33
Regions
GpiFloodFill 4-51
GpiFrameRegion 4-51
Searching Directories
DosFindClose 4-32
DosFindFirst 4-32
DosFindNext 4-32
Semaphores
DosAddMuxWaitSem 4-19
DosCloseEventSem 4-19
DosCloseMutexSem 4-19
DosCloseMuxWaitSem 4-19
DosCloseSem 4-19
DosCreateEventSem 4-19
DosCreateMutexSem 4-19
DosCreateMuxWaitSem 4-19
DosCreateSem 4-19
DosDeleteMuxWaitSem 4-19
DosFSRamSemClear 4-19
DosFSRamSemRequest 4-19
DosMuxSemWait 4-19
DosOpenEventSem 4-19
DosOpenMutexSem 4-19
DosOpenMuxWaitSem 4-19
DosOpenSem 4-19
DosPostEventSem 4-19
DosQueryEventSem 4-19
DosQueryMutexSem 4-19
DosQueryMuxWaitSem 4-19
DosReleaseMutexSem 4-19
DosRequestMutexSem 4-19
DosResetEventSem 4-19
DosSemClear 4-19
DosSemRequest 4-19
DosSemSet 4-19
DosSemSetWait 4-19
DosSemWait 4-19
DosWaitEventSem 4-19
DosWaitMuxWaitSem 4-19
Session Management
DosSelectSession 4-35
DosSetSession 4-35
DosSMRegisterDD 4-35
DosStartSession 4-35
DosStopSession 4-35
Setting Available Number of File Handles
DosSetMaxFH 4-34
DosSetRelMaxFH 4-34
Setting Memory Commitment and Access
DosSetMem 4-11

Functions *(continued)*
Setting the File Mode
DosSetFileMode 4-33
Signalling Events with Semaphores
DosCloseEventSem 4-20
DosCreateEventSem 4-20
DosOpenEventSem 4-20
DosPostEventSem 4-20
DosQueryEventSem 4-20
DosResetEventSem 4-20
DosWaitEventSem 4-20
Signals
DosHoldSignal 4-35
DosSetSigHandler 4-35
Spooler
DosPrintDestAdd 4-39
DosPrintDestControl 4-39
DosPrintDestDel 4-39
DosPrintDestEnum 4-39
DosPrintDestGetInfo 4-39
DosPrintDestSetInfo 4-39
DosPrintDriverEnum 4-39
DosPrintJobContinue 4-39
DosPrintJobDel 4-39
DosPrintJobEnum 4-39
DosPrintJobGetId 4-39
DosPrintJobGetInfo 4-39
DosPrintJobPause 4-39
DosPrintJobSetInfo 4-39
DosPrintPortEnum 4-39
DosPrintQAdd 4-39
DosPrintQContinue 4-39
DosPrintQDel 4-39
DosPrintQEnum 4-39
DosPrintQGetInfo 4-39
DosPrintQPause 4-39
DosPrintQProcessorEnum 4-39
DosPrintQPurge 4-39
DosPrintQSetInfo 4-39
SplControlDevice 4-39
SplCreateDevice 4-39
SplCreateQueue 4-39
SplDeleteDevice 4-39
SplDeleteJob 4-39
SplDeleteQueue 4-39
SplEnumDevice 4-39
SplEnumDriver 4-39
SplEnumJob 4-39
SplEnumPort 4-39
SplEnumQueue 4-39
SplEnumQueueProcessor 4-39
SplHoldJob 4-39
SplHoldQueue 4-39
SplPurgeQueue 4-39
SplQueryDevice 4-39
SplQueryJob 4-39
SplQueryJobId 4-39
SplQueryQueue 4-39
SplReleaseJob 4-39

Functions *(continued)*
Spooler *(continued)*
SplReleaseQueue 4-39
SplSetDevice 4-39
SplSetJobInfo 4-39
SplSetQueue 4-39
Starting Programs
DosExecPgm 4-16
Transformations
GpiConvertWithMatrix 4-52
Using Named Pipes
DosCallNmPipe 4-27
DosCallNPipe 4-27
DosConnectNmPipe 4-27
DosConnectNPipe 4-27
DosCreateNmPipe 4-27
DosCreateNPipe 4-27
DosDisConnectNmPipe 4-27
DosDisConnectNPipe 4-27
DosPeekNmPipe 4-27
DosPeekNPipe 4-27
DosQNmPHandState 4-27
DosQueryNmPipeInfo 4-27
DosQueryNmPipeSemState 4-27
DosQueryNPHState 4-27
DosQueryNPipeInfo 4-27
DosQueryNPipeSemState 4-27
DosRawReadNmPipe 4-27
DosRawWriteNmPipe 4-27
DosSetNmHandInfo 4-27
DosSetNmPipeSem 4-27
DosSetNPHState 4-27
DosSetNPipeSem 4-27
DosTransactNmPipe 4-27
DosTransactNPipe 4-27
DosWaitNmPipe 4-27
DosWaitNPipe 4-27
Using Named Shared Memory
DosAllocSharedMem 4-6
DosAllocShrSeg 4-6
DosGetNamedSharedMem 4-6
DosGetShrSeg 4-6
Using Queues
DosCloseQueue 4-28
DosCreateQueue 4-28
DosOpenQueue 4-28
DosPeekQueue 4-28
DosPurgeQueue 4-28
DosQueryQueue 4-28
DosReadQueue 4-28
DosWriteQueue 4-28
Using Semaphores for Multiple Waiting
DosAddMuxWaitSem 4-25
DosCloseMuxWaitSem 4-25
DosCreateMuxWaitSem 4-25
DosDeleteMuxWaitSem 4-25
DosOpenMuxWaitSem 4-25
DosQueryMuxWaitSem 4-25
DosWaitMuxWaitSem 4-25

Functions *(continued)*
Using Semaphores for Mutual Exclusion
DosCloseMutexSem 4-21
DosCreateMutexSem 4-21
DosOpenMutexSem 4-21
DosQueryMutexSem 4-21
DosReleaseMutexSem 4-21
DosRequestMutexSem 4-21
Using Threads and Processes
DosCallBack 4-12
DosCreateThread 4-12
DosCWait 4-12
DosDebug 4-12
DosEnterCritSec 4-12
DosExecPgm 4-12
DosExit 4-12
DosExitCritSec 4-12
DosExitList 4-12
DosGetEnv 4-12
DosGetInfoBlocks 4-12
DosGetInfoSeg 4-12
DosGetPID 4-12
DosGetPPID 4-12
DosGetPrty 4-12
DosKillProcess 4-12
DosPTrace 4-12
DosQSysInfo 4-12
DosResumeThread 4-12
DosRetForward 4-12
DosR2StackRealloc 4-12
DosSetPriority 4-12
DosSetPrty 4-12
DosSuspendThread 4-12
DosWaitChild 4-12
DosWaitThread 4-12
Using Timers
DosAsyncTimer 4-28
DosGetDateTime 4-28
DosSetDateTime 4-28
DosSleep 4-28
DosStartTimer 4-28
DosStopTimer 4-28
DosTimerAsync 4-28
DosTimerStart 4-28
DosTimerStop 4-28
Using Unnamed Pipes
DosCreatePipe 4-26
DosMakePipe 4-26
Using Unnamed Shared Memory
DosGetSeg 4-8
DosGetSharedMem 4-8
DosGiveSeg 4-8
DosGiveSharedMem 4-8
VDD Services
DosCloseVDD 4-37
DosOpenVDD 4-37
DosRequestVDD 4-37
Waiting for Threads
DosCWait 4-14

Functions *(continued)*
Waiting for Threads *(continued)*
DosWaitChild 4-14
DosWaitThread 4-14
WinCheckButton 4-53
WinCheckMenuItem 4-53
WinCreateObject 8-40
WinDeleteLboxItem 4-53
WinDeregisterObjectClass 8-40
Window Locking
WinLockWindow 4-38
WinLockWindowUpdate 4-38
WinQueryWindowLockCount 4-38
Window Management
WinRegisterWinDestroy 4-38
WinEnableControl 4-53
WinEnableMenuItem 4-53
WinEnumObjectClasses 8-40
WinFreeIcon 8-40
WinInsertLboxItem 4-53
WinIsControlEnabled 4-53
WinIsMenuItemChecked 4-53
WinIsMenuItemValid 4-53
WinLoadFileIcon 8-40
WinMenuItemEnabled 4-53
WinQueryButtonCheckState 4-53
WinQueryLboxCount 4-53
WinQueryLboxItemText 4-53
WinQueryLboxItemTextLength 4-53
WinQueryLboxSelectedItem 4-53
WinQueryObject 8-40
WinReplaceObjectClass 8-40
WinRestoreWindowPos 8-40
WinSetFileIcon 8-40
WinSetLboxItemText 4-53
WinSetMenuItemText 4-53
WinSetObjectData 8-40
WinShutdownSystem 8-40
WinStoreWindowPos 8-40
Workplace
WinCreateObject 4-40
WinDeregisterObjectClass 4-40
WinDestroyObject 4-40
WinEnumObjectClasses 4-40
WinFreeFileIcon 4-40
WinLoadFileIcon 4-40
WinRegisterObjectClass 4-40
WinReplaceObjectClass 4-40
WinRestoreWindowPos 4-40
WinSetFileIcon 4-40
WinSetObjectData 4-40
WinShutdownSystem 4-40
WinStoreWindowPos 4-40
Writing Asynchronously
DosWriteAsync 4-34
functions, naming guidelines 4-1

# G

gate 2-9
descriptor 2-9, 2-10
types of 2-10
types of 2-9
GetData 7-15, 7-17
get_ 7-17
global data, DLLs 5-7
Global Descriptor Table Register (GDTR) 2-3
Global Descriptor Table (GDT) 2-5, 3-15
GpiCharString 1-27
GpiCharStringPos 1-27
GpiConvertWithMatrix 4-52
GpiCreateLogFont 1-28
GpiDrawBits 4-51
GpiDrawSegment 1-28
GpiFloodFill 4-51
GpiFrameRegion 4-51
GpiLoadPublicFonts 4-52
GpiPathToRegion 4-51
GpiPolylineDisjoint 4-52
GpiQueryCharBreakExtra 4-52
GpiQueryCharExtra 4-52
GpiQueryCharOutline 4-52
GpiQueryFaceString 4-52
GpiQueryLogicalFont 4-52
GpiSetAttrs 1-25
GpiSetCharBreakExtra 4-52
GpiSetCharExtra 4-52
GpiSetCharSet 1-28
GpiSetColor 1-25
GpiUnloadPublicFonts 4-52
granularity 4-3
granularity bit 2-5
graphic primitives 1-25
arc 1-25
area 1-25
image 1-25
line 1-25
marker 1-25
graphic tools 1-26, 1-27
bit map 1-26
clipping 1-27
font 1-26
logical color palette 1-26
path 1-26
transformation 1-27
Graphics Programming Interface 1-24
functions 1-30
groups, SOM methods 7-33
group, methods 7-8
guard page attribute 2-17

# H

hard-copy output 1-28

hardware interrupt handler routines 6-16
header file, SOM.H 7-18
heap 4-4
help information 4-40
  developing 1-24
  multiple viewport 4-40
help panel 1-24
help view of an object 8-3
helper macros 4-52
HELPPANEL 8-32
high performance file system (HPFS) 1-8
HIMEM area 6-10
hooks 4-51
hot spot 1-19

# I

IBM System Object Model (SOM) 8-1, 8-8
Icon Editor 1-22
ICONFILE 8-32
ICONPOS 8-33
ICONRESOURCE 8-33
icons 1-21
icons view of an object 8-2
icons, user interface 8-2
ID manipulation, SOM 7-18
idle-time threads 1-7
image 1-25
implementation header file for SOM class, .IH 7-12
IMPLIB, import library 5-12
implied metaclass 7-30
import libraries 5-10
import library, contents 5-10
IMPORTS statement, module-definition file 5-10
in-use list, Workplace objects 8-30
include file layout 3-10
include section, class definition file 7-7
indirect manipulation 8-4
Information Presentation Facility (IPF) 1-23, 4-40, 8-21
inheritance 7-4, 7-24
inheritance, object characteristics and behavior 8-3
INITGLOBAL, LIBRARY statement 5-13
initialization, SOM 7-5
INITINSTANCE, LIBRARY statement 5-13
input message 1-3
input/output processing 2-11
installation, Workplace objects
  installation batch files, REXX 8-44
  installation programs 8-42
  Workplace Class List Object 8-45
instance data 2-15, 3-4, 7-7, 7-21
instance data, DLL 5-7
instance method 7-4
instance of class 7-4
instruction decode unit 2-2
instructions, pipelined execution of 2-2
INT 21h multiplex 6-12
INT 31h services 6-11
INT 32h interface 6-12
interaction with objects
  direct manipulation, drag/drop 8-4
  indirect manipulation 8-4
interprocess communication 1-10
  protocols 1-10
    device support 1-12
    exception handling 1-11
    multiple DOS sessions 1-11
    pipes 1-10
    queues 1-11
    semaphores 1-10
    shared memory 1-11
Interrupt Descriptor Table Register (IDTR) 2-3
interrupt gates 2-10
invoking methods, SOM 7-20
IPF tags
  :acviewport 4-41
  :br 4-41
  :ddf 4-41
  :docprof 4-41
  :font 4-41
  :table 4-41
  :title 4-41
item types for in-use list
I/O address space 2-11
  protection mechanisms 2-11

# K

keyboard 6-15
keyboard input 1-19
Keyboard object 8-7
KEYNAMES, object setup variables 8-34
KEYNAME, object setup variables 8-32, 8-41
KILLPROCESS 4-36

# L

language binding files, SOM 7-6
language binding files, SOM compiler
  C program template for class implementation 7-12
  .CS2, stylized form of CSC 7-12
  .C,C program template for class
    implementation 7-12
  .DEF, linker file 7-12
  .H, public header file for SOM client 7-12
  .IH, implementation header file 7-12
  .PH, private implementation header file 7-12
  .PSC, private class implementation file 7-12
  .SC, public class definition file 7-12
LDT descriptors, used by 3-17
LDT tiling 2-18
LIBPATH, locating DLLs 5-5
library files 3-9
LIBRARY statement options
  INITGLOBAL 5-9, 5-13

LIBRARY statement options *(continued)*
INITINSTANCE 5-9, 5-13
TERMGLOBAL 5-9
TERMINSTANCE 5-9
library support 3-12
LIM EMS emulation 6-7
LIM EMS emulation, implementation of 6-7
LIMA XMS emulation 6-9
limit checking 2-8
line 1-25
linear address 1-11
linear memory model 1-1
load option, LOADONCALL 5-6
load option, PRELOAD 5-6
load-time dynamic linking 5-3
loading a DLL 5-6
LOADONCALL, defining CODE attributes 5-6
Local Descriptor Table Register (LDTR) 2-3
Local Descriptor Table (LDT) 2-5, 2-14
locating DLLs, LIBPATH 5-5
LOCK prefix instructions 2-13
logical color palette 1-26
logical devices 1-5
logical view of storage media 1-8

# M

main thread 1-6
main window 1-14
MakeMyWindow() registration routine 3-29
marker 1-25
MEMMAN statement, CONFIG.SYS 5-20
memory allocation 4-2
memory commitment and access 4-11
memory extender support 6-6
memory extenders
LIMA XMS Version 2.0 6-8
memory management 1-7
Memory Management Methods 8-9
Memory Management, SOM 7-19
memory objects 1-11, 2-15, 4-1
characteristics of 2-15
discarding 4-11
size limit 4-1
memory overcommitment feature 4-11
memory segments 1-7
maximum size 1-7
memory suballocation 4-4
MEMORYITEM 8-30
menu 1-21
menu bar 1-17
message input parameters 1-21
message processing 1-19
message queue 3-3
message queues 1-11
message retrieval functions 4-34
messages 1-19
metaclass 7-28
metaclass section, class definition file 7-7
metaclasses 7-5
metafiles 1-28, 4-40
definition of 1-28
method 7-3
method groups 7-8
method groups, SOM 7-33
method groups, Workplace 8-9
method overrides 7-4
method stub, SOM 7-14
MethodDebug 7-15, 7-17
methods section, class definition file 7-7
migrating to OS/2 2.0 3-33
migration, 32-Bit 4-41
MINWIN 8-34
mixed 16-bit applications 3-6
mixed 32-bit applications 3-8
modal dialog box 1-18
modeless dialog box 1-18
module-definition file 5-6, 5-10
mouse 6-15
mouse capture window 1-19
mouse input 1-19
Mouse object 8-7
mouse pointer 1-19
MPFROMP macro 3-30
multiple DOS Mode session, creating 6-2
multiple DOS sessions 1-11
multiple viewport, help information 4-40
multitasking
definition of 1-4
levels 1-5
types of 1-4
mutex semaphores 1-10
muxwait semaphores 1-10

# N

name shared memory 1-11
named pipes 1-10
named shared memory 4-6
naming conventions, SOM 7-16
New 7-29
NewClass 7-23, 7-29
NewClass, SOM function 7-17
New, SOM Macro 7-17
NOCOPY 8-33
NODELETE 8-32
NODRAG 8-34
NOMOVE 8-33
non-persistent objects 8-9
NOPRINT 8-33
NORENAME 8-34
NOSHADOW 8-33
notebook window control 4-44
messages
BKM_CALCPAGERECT 4-45
BKM_DELETEPAGE 4-45

notebook window control *(continued)*
messages *(continued)*
BKM_INSERTPAGE 4-45
BKM_INVALIDATETABS 4-45
BKM_QUERYPAGECOUNT 4-45
BKM_QUERYPAGEID 4-45
BKM_QUERYPAGEULONG 4-45
BKM_QUERYPAGEWINDOWHWND 4-45
BKM_QUERYTABBITMAP 4-45
BKM_QUERYTABTEXT 4-45
BKM_SETDIMENSIONS 4-45
BKM_SETPAGEULONG 4-45
BKM_SETPAGEWINDOWHWND 4-45
BKM_SETSTATUSLINETEXT 4-45
BKM_SETTABBITMAP 4-45
BKM_SETTABTEXT 4-45
BKM_TURNTOPAGE 4-45
BKN_NEWPAGESIZE 4-45
BKN_PAGESELECTED 4-45
styles
BKS_BACKPAGESBL 4-44
BKS_BACKPAGESBR 4-44
BKS_BACKPAGESTL 4-44
BKS_BACKPAGESTR 4-44
BKS_MAJORTABBOTTOM 4-45
BKS_MAJORTABLEFT 4-45
BKS_MAJORTABRIGHT 4-45
BKS_MAJORTABTOP 4-45
BKS_POLYGONTABS 4-45
BKS_ROUNDEDTABS 4-45
BKS_SQUARETABS 4-45
BKS_STATUSTEXTCENTER 4-45
BKS_STATUSTEXTLEFT 4-45
BKS_STATUSTEXTRIGHT 4-45
NOTVISIBLE 8-33
numeric coprocessor (80287) 2-12
numeric coprocessor (80387) 2-12, 6-15

# O

object 7-3
object characteristics and behavior, inheritance 8-3
Object Class Styles
OBJSTYLE_NOCOPY 8-22
OBJSTYLE_NODELETE 8-22
OBJSTYLE_NODRAG 8-22
OBJSTYLE_NOMOVE 8-22
OBJSTYLE_NOPRINT 8-22
OBJSTYLE_NORENAME 8-22
OBJSTYLE_NOSHADOW 8-22
OBJSTYLE_NOTDEFAULTICON 8-22
OBJSTYLE_NOTVISIBLE 8-22
OBJSTYLE_TEMPLATE 8-22
object details, Workplace
wpclsQueryDetailsInfo 8-23
wpQueryDetailsData 8-23
Object IDs for System Folders
<WP_CONFIG> 8-41
Object IDs for System Folders *(continued)*
<WP_DESKTOP> 8-41
<WP_DRIVE> 8-41
<WP_INFO> 8-41
<WP_NOWHERE> 8-41
<WP_START> 8-41
<WP_SYSTEM> 8-41
<WP_TEMPS> 8-41
Object Information Methods
wpSet/wpQueryDefaultHelp 8-22
wpSet/wpQueryIcon 8-22
wpSet/wpQueryIconData 8-22
wpSet/wpQueryStyle 8-22
wpSet/wpQueryTitle 8-22
object instance 7-4
Object Interface Definition Language 7-6, 8-8
Object Interface Definition Language, OIDL 7-6
object rendering
OBJECT 8-28
OS2FILE 8-28
object templates, Workplace 8-22
Object Usage Methods
wpAddToObjectUseList 8-30
wpDeleteFromObjUseList 8-30
wpFindUseItem 8-30
object view 1-18
object-oriented user environments 8-1
object-oriented user interface, designing 8-4
OBJECTID 8-33
objects, parts of 8-9
objects, user interface 8-2
OBJSTYLE_NOCOPY 8-22
OBJSTYLE_NODELETE 8-22
OBJSTYLE_NODRAG 8-22
OBJSTYLE_NOMOVE 8-22
OBJSTYLE_NOPRINT 8-22
OBJSTYLE_NORENAME 8-22
OBJSTYLE_NOSHADOW 8-22
OBJSTYLE_NOTDEFAULTICON 8-22
OBJSTYLE_NOTVISIBLE 8-22
OBJSTYLE_TEMPLATE 8-22
OIDL, Object Interface Definition Language 7-6
OIDL, SOM Object Interface Definition Language 7-6
on-chip caching 2-2
OPEN 8-34
Open Actions 8-20
Open dialog 4-42
Open Views 8-20
Open Views for Workplace Objects
OPEN_CONTENTS 8-20
OPEN_DEFAULT 8-20
OPEN_DETAILS 8-20
OPEN_HELP 8-20
OPEN_RUNNING 8-20
OPEN_SETTINGS 8-20
OPEN_TREE 8-20
OPEN_USER 8-20

OPEN_CONTENTS 8-20
OPEN_DEFAULT 8-20
OPEN_DETAILS 8-20
OPEN_HELP 8-20
OPEN_RUNNING 8-20
OPEN_SETTINGS 8-20
OPEN_TREE 8-20
OPEN_USER 8-20
OPEN_, Open Views for Workplace Objects 8-20
OS2286.LIB 3-9
OS2386.LIB 3-9, 5-10
OS2.H include file 3-10
OS/2 and the 80386 processor
  compatibility with 16-bit OS/2 2-18
  memory objects and memory sharing 2-15
  page attributes and memory access protection 2-17
  process address space 2-14
OS/2 base memory, available amount 6-6
OS/2 scheduler 6-1
OS/2 system functions 1-29
OS/2 system functions, groups 1-29
OS/2 2.0
  compatibility with version 1.X 1-2
  Enhanced DOS Session 1-2
  multiple DOS sessions 1-2
  overview 1-1
  portability 1-1
  virtual device drivers 1-2
  386 features 1-1
override methods 7-4
owner window 1-15
OWNERDRAW 8-23

# P

page 2-7
Page Directory table 2-7
page specifiers 2-7
Page Table 2-7
  page specifiers 2-7
  shared pages, defining 2-7
pageable virtual memory 6-10
paged environment 1-12
paged memory model
  caching mechanism 2-7
  dedicated paging unit 2-7
  making memory references 2-7
  page 2-7
  Page Directory table 2-7
  Page Table 2-7
  performance implications 2-6
paged virtual memory 1-1
page, attributes 2-17
paging unit 2-2
PAG_COMMIT 4-29
PAG_EXECUTE 4-9
PAG_READ 4-29
PAG_WRITE 4-9
parent class 7-4, 8-3
parent class section, class definition file 7-7
parent process 1-6
parent_ 7-17
passthru section, class definition file 7-7
path 1-26
paths 4-51
peer threads 1-6
persistent objects 8-9
physical device drivers (PDD) 6-2, 6-14
physical devices 1-5
pipe handles
  read 1-10
  write 1-10
pipes 1-10
  named 1-10
  unnamed 1-10
plotter presentation drivers 1-24
PM
  application
  application template 3-3
  applications 1-3, 3-2, 3-3
  device functions 1-29
  features of 1-13
  structure of 3-3
PMWIN.H 4-52
pointer 1-22
Polylines 4-52
polymorphism 7-4, 7-14
Pop-up Menu Methods
  wpClose 8-20
  wpCopyObject 8-20
  wpCreateFromTemplate 8-20
  wpCreateShadowObject 8-20
  wpDelete 8-20
  wpDisplayHelp 8-20
  wpFilterPopupMenu 8-13
  wpHide 8-20
  wpInsertPopupMenuItems 8-13
  wpMenuItemHelpSelected 8-19
  wpMenuItemSelected 8-19
  wpModifyPopupMenu 8-13
  wpMoveObject 8-20
  wpOpen 8-20
  wpPrintObject 8-20
  wpRestore 8-20
pop-up menus, parts of 8-12
pre-emptive multitasking 1-12
predefined Workplace objects
PRELOAD, defining CODE attributes 5-6
preregistered window classes 1-15
presentation drivers 1-24
  definition of 1-24
Presentation Manager API
  WinSetWindowThunkProc() 3-31

presentation spaces 1-24
  definition of 1-24
PrfAddProgram 4-37
PrfChangeProgram 4-37
PrfCreateGroup 4-37
PrfDestroyGroup 4-37
PrfQueryDefinition 4-37
PrfQueryProgramCategory 4-37
PrfQueryProgramHandle 4-37
PrfQueryProgramTitles 4-37
PrfRemoveProgram 4-37
primary windows 1-18
primitive attributes 1-25
printer 6-15
Printer object 8-7
printer presentation drivers 1-24
printing 4-38
printing in the Workplace Shell 8-46
private class implementation file,.PSC 7-12
private implementation header file for SOM class, .PH 7-12
private interface, SOM classes 7-12
private memory descriptors 3-17
private methods 7-12
private semaphores 4-17
private window class 1-15
privilege levels 2-9
  level 0 2-9
  level 1 2-9
  level 2 2-9
  level 3 2-9
procedure entry points, restrictions 2-9
Process Information Block (PIB) 4-14
process virtual memory map 4-11
processes 1-6, 4-14
  obtaining information about 4-14
processes, definition of 1-6
profile functions 1-30
program object 1-15
Programmable Interrupt Controller 6-15
programming models
  mixed 16-bit 3-5
  mixed 32-bit 3-5
  pure 16-bit 3-5
  pure 32-bit 3-5
protect DLLs, MEMMAN statement 5-20
protect mode 2-3, 6-1
protect mode program loading 6-12
protected memory use 5-19
protected system environment 6-1
public class implementation file, .SC 7-12
public header file for SOM client, .H 7-12
public header file, .H 7-23
public interface, SOM classes 7-12
public window class 1-15
pull-down menus 8-12
pure 16-bit applications 3-5
pure 32-bit applications 3-7

## Q

querying file mode 4-33
querying system information 4-33
queue 1-11
  access
    first in, first out (FIFO) 1-11
    last in, first out (LIFO) 1-11
  definition 1-11
queued Input 1-3
Queues 4-27

## R

random-access memory (RAM) 6-10
read-only memory (ROM) 6-8
reading files asynchronously 4-33
real mode 2-3
  memory references 2-4
  physical memory addresses 2-3
  program addresses 2-3
  virtual memory, support 2-3
real mode execution 6-4
real mode software 6-11
  execution of 6-11
real mode task 6-1
RECORDITEM 8-30
record, object details 8-23
reflections (CUA), shadows (Workplace) 8-21
regions 4-51
registering a view of an object 8-38
regular threads 1-7
release order section, class definition file 7-7, 7-8
Renew, SOM Macro 7-17
Replaceable SOM functions
  SOMCalloc 7-19
  SOMClassInitFuncName 7-19
  SOMDeleteModule 7-19
  SOMError 7-19
  SOMFree 7-19
  SOMLoadModule 7-19
  SOMMalloc 7-19
  SOMOutCharRoutine 7-18, 7-19
  SOMRealloc 7-19
reserved instructions 2-10
resource compiler 1-22
Resource Editors 1-22
resource file 1-21
  definition of 1-21
resources
  accelerator table 1-21
  bit map 1-21
  dialog includes 1-21
  dialog templates 1-21
  fonts 1-21
  icons 1-21

resources *(continued)*
menu 1-21
pointer 1-22
string tables 1-22
types of 1-21
window templates 1-21
Restrictions 6-9
retained segments 1-28
REXX Utility Workplace Functions
SysCreateObject 8-42
SysDeregisterObjectClass 8-42
SysQueryClassList 8-42
SysRegisterObjectClass 8-42
REXX, object installation 8-44
ring 3 segment 2-14
root directory 1-8
run-time dynamic linking 5-6

## S

SaveAs dialog 4-42
Save/Restore State Methods
wpSaveDeferred 8-29
wpSaveImmediate 8-29
wpSave/wpRestoreData 8-29
wpSave/wpRestoreLong 8-29
wpSave/wpRestoreState 8-29
wpSave/wpRestoreString 8-29
Scheme Palette object 8-7
scroll bars 1-17
seamless environment 8-6
secondary windows 1-18
segment selector 2-5
using as index to descriptor table 2-5
using to specify privilege level 2-5
segment swapping 4-1
segmentation unit 2-2
segmented dynamic link model 3-12
segmented memory model 1-1, 4-1
address translation 2-5
descriptor table 2-5
logical address space 2-4
segment selector 2-5
segmented pointers, use by flat pointers 3-11
segments, size limit 4-1
segment:offset address format 6-1
SEG_DISCARDABLE 4-11
selector:offset address format 6-1
semaphores 1-10, 4-17
definition 1-10
event 1-10, 4-17
fast-safe RAM 4-17
mutex 1-10, 4-17
muxwait 1-10, 4-17
ownership 4-21
private semaphores 4-17
rules for use 4-18
shared semaphores 4-17
sending output
to hard-copy device 1-28
to spooler 1-28
server process 1-10
session management 4-35
sessions 1-2, 1-5
definition of 1-5
hierarchy 1-6
setting file mode 4-33
settings notebook
adding pages 8-11
methods
wpAddObjectGeneralPage 8-11
wpAddSettingsPages 8-11
wpInsertSettingsPage 8-11
removing pages 8-12
settings view 8-3
Settings-Notebook Methods 8-9, 8-10
setup variables, KEYNAME 8-41
Setup/Cleanup Methods
wpFree 8-32
wpInitData 8-32
wpScanSetupString 8-32
wpSetup 8-32
wpUnInitData 8-32
Set/Cleanup Methods 8-9
Set/Query Object Information Methods 8-9
shadow of an object 8-2
shadows (Workplace), reflections (CUA) 8-21
shared data, DLLs 5-7
shared memory 1-11
shared memory descriptors 3-17
shared resources 1-4
shared semaphores 4-17
Shredder object 8-7
sibling windows 1-14
signal functions 4-35
signals
CTRL+BREAK 4-35, 4-36
CTRL+C 4-35, 4-36
KILLPROCESS 4-36
simple value parameters, passing 3-30
slider window control 4-49
messages
SLM_ADDETENT 4-50
SLM_QUERYDETENTPOS 4-50
SLM_QUERYSCALETEXT 4-50
SLM_QUERYSLIDERINFO 4-50
SLM_QUERYTICKPOS 4-50
SLM_QUERYTICKSIZE 4-50
SLM_REMOVEDETENT 4-50
SLM_SETSCALETEXT 4-50
SLM_SETSLIDERINFO 4-50
SLM_SETTICKSIZE 4-50
SLN_CHANGE 4-50
SLN_KILLFOCUS 4-50
SLN_SETFOCUS 4-50
SLN_SLIDERTRACK 4-50

slider window control *(continued)*
styles
SLS_BOTTOM 4-50
SLS_BUTTONSBOTTOM 4-50
SLS_BUTTONSLEFT 4-50
SLS_BUTTONSRIGHT 4-50
SLS_BUTTONSTOP 4-50
SLS_CENTER 4-50
SLS_HOMEBOTTOM 4-50
SLS_HOMELEFT 4-50
SLS_HOMERIGHT 4-50
SLS_HOMETOP 4-50
SLS_HORIZONTAL 4-49
SLS_LEFT 4-50
SLS_OWNERDRAW 4-50
SLS_PRIMARYSCALE1 4-50
SLS_PRIMARYSCALE2 4-50
SLS_READONLY 4-50
SLS_RIBBONSTRIP 4-50
SLS_RIGHT 4-50
SLS_SNAPTOINCREMENT 4-50
SLS_TOP 4-50
SLS_VERTICAL 4-49
SLM_ADDETENT 4-50
SLM_QUERYDETENTPOS 4-50
SLM_QUERYSCALETEXT 4-50
SLM_QUERYSLIDERINFO 4-50
SLM_QUERYTICKPOS 4-50
SLM_QUERYTICKSIZE 4-50
SLM_REMOVEDETENT 4-50
SLM_SETSCALETEXT 4-50
SLM_SETSLIDERINFO 4-50
SLM_SETTICKSIZE 4-50
SLN_CHANGE 4-50
SLN_KILLFOCUS 4-50
SLN_SETFOCUS 4-50
SLN_SLIDERTRACK 4-50
SLS_BOTTOM 4-50
SLS_BUTTONSBOTTOM 4-50
SLS_BUTTONSLEFT 4-50
SLS_BUTTONSRIGHT 4-50
SLS_BUTTONSTOP 4-50
SLS_CENTER 4-50
SLS_HOMEBOTTOM 4-50
SLS_HOMELEFT 4-50
SLS_HOMERIGHT 4-50
SLS_HOMETOP 4-50
SLS_HORIZONTAL 4-49
SLS_LEFT 4-50
SLS_OWNERDRAW 4-50
SLS_PRIMARYSCALE1 4-50
SLS_PRIMARYSCALE2 4-50
SLS_READONLY 4-50
SLS_RIBBONSTRIP 4-50
SLS_RIGHT 4-50
SLS_SNAPTOINCREMENT 4-50
SLS_TOP 4-50
SLS_VERTICAL 4-49
SMEMIT, SOM environment variable 7-12
SMEMIT, SOM Environment Variables 7-13
SMINCLUDE, SOM environment variable 7-12
SMINCLUDE, SOM Environment Variables 7-13
SMTMP, SOM environment variable 7-12
SMTMP, SOM Environment variables 7-13
SOM class libraries 7-32
SOM compiler 7-6, 7-11, 8-8
SOM Debug Control Variables
SOM_AssertLevel 7-18
SOM_WarnLevel 7-18
SOM Debug Macros
SOM_Assert 7-18
SOM_Expect 7-18
SOM_TestC 7-18
SOM_WarnMsg 7-18
SOM environment variables
SMEMIT 7-12, 7-13
SMINCLUDE 7-12, 7-13
SMTMP 7-12, 7-13
SOM Error Severity Levels
SOM_Fatal 7-19
SOM_Ignore 7-19
SOM_Warn 7-19
SOM function
NewClass 7-17
somPrintf 7-23
SOM ID Manipulation Functions
somBeginPersistentIds 7-18
somEndPersistentIds 7-18
somRegisterID 7-18
somSetExpectedIDs 7-18
somTotalRegIDs 7-18
somUniqueKey 7-18
SOM ID Manipulation Macros
SOM_CheckID 7-18
SOM_CompareIDs 7-18
SOM_IDFromString 7-18
SOM_StringFromID 7-18
SOM IDs 7-18
SOM initialization 7-5
SOM macros
access instance data 7-17
GetData 7-15, 7-17
get_ 7-17
MethodDebug 7-15, 7-17
New 7-17
parent_ 7-17
Renew 7-17
SOM_ERROR 7-19
SOM_GetClass 7-19
SOM_ParentResolve 7-17
SOM_Resolve 7-17
SOM_ResolveNoCheck 7-17
SOM_TEST 7-19
underscore macro 7-16

SOM naming conventions 7-16
SOM run-time environment
  SOMClass 7-5
  SOMClassMgr 7-5
  SOMObject 7-5
SOM Tracing Facility 7-15
somBeginPersistentIds, SOM ID manipulation 7-18
SOMCalloc 7-19
SOMClass 7-5, 7-28
SOMClassInitFuncName 7-19
SOMClassMgr 7-5
SOMClassMgrObject 7-5
SOMDeleteModule 7-19
somEndPersistentIds, SOM ID manipulation 7-18
SOMError 7-19
SOMFree 7-19, 7-24
SOMLINK 7-14
SOMLoadModule 7-19
SOMMalloc 7-19
somNew 7-23
SOMObject 7-5, 7-13, 7-28, 7-32, 8-8
SOMOutCharRoutine 7-19
SOMOutCharRoutine, replaceable SOM function 7-18
somPrintf 7-19, 7-23
SOMRealloc 7-19
somRegisterId, SOM ID manipulation 7-18
somSelf 7-14
somSetExpectedIds, SOM ID manipulation 7-18
somThis 7-15
somTotalRegIds, SOM ID manipulation 7-18
somUniqueKey, SOM ID manipulation 7-18
SOM.H, SOM header file 7-18
SOM_Assert 7-18
SOM_AssertLevel 7-18
SOM_CheckID, SOM ID manipulation 7-18
SOM_CompareIDs, SOM ID manipulation 7-18
SOM_CurrentClass 7-31
SOM_ERROR 7-19
SOM_Expect 7-18
SOM_Fatal 7-19
SOM_GetClass 7-19
SOM_IDFromString, SOM ID manipulation 7-18
SOM_Ignore 7-19
SOM_ParentResolve 7-17
SOM_Resolve 7-17
SOM_ResolveNoCheck 7-17
SOM_Scope 7-14
SOM_StringFromID, SOM ID manipulation 7-18
SOM_TEST 7-19
SOM_TestC 7-18
SOM_Warn 7-19
SOM_WarnLevel 7-18
SOM_WarnMsg 7-18
Sound object 8-7
source object 8-4
source window 1-23
Special Needs object 8-7
SplControlDevice 4-39
SplCreateDevice 4-39
SplCreateQueue 4-39
SplDeleteDevice 4-39
SplDeleteJob 4-39
SplDeleteQueue 4-39
SplEnumDevice 4-39
SplEnumDriver 4-39
SplEnumJob 4-39
SplEnumPort 4-39
SplEnumQueue 4-39
SplEnumQueueProcessor 4-39
SplHoldJob 4-39
SplHoldQueue 4-39
SplPurgeQueue 4-39
SplQueryDevice 4-39
SplQueryJob 4-39
SplQueryJobId 4-39
SplQueryQueue 4-39
SplReleaseJob 4-39
SplReleaseQueue 4-39
SplSetDevice 4-39
SplSetJobInfo 4-39
SplSetQueue 4-39
spooler functions 1-30
Spooler object 8-7
standard dialog boxes 4-42
standard window elements 1-16
standard windows 1-16
standard-error file 1-9
standard-input file 1-9
standard-output file 1-9
static linking 5-1
static linking, advantages 5-2
static linking, disadvantages 5-2
storage classes 8-9
string tables 1-22
stylized form of CSC, .CS2 7-12
subclass 7-24, 8-3
subclasses 7-4
subclassing SOM classes 7-12
subclassing windows 1-16
subdirectory 1-8
supported 16-bit subsystems 3-10
swap space 1-8, 4-1
SysCreateObject 8-42, 8-44
SysDeregisterObjectClass 8-42, 8-44
SysQueryClassList 8-42
SysRegisterObjectClass 8-42, 8-44
system information, querying 4-33
System object 8-7
system queue 1-20

## T

target object 8-4
target window 1-23

task protection 2-8
  types of 2-8
Task Register (TR) 2-3
TEMPLATE 8-32
Templates Folder 8-44
Templates folder, Workplace 8-23
templates, Workplace objects 8-22
Thread Information Block (TIB) 4-14
thread 1 1-6
threads 4-12, 4-13, 4-14
  controlling 4-13
  creating threads 4-12
  definition of 1-6
  exiting from 4-13
  obtaining information about 4-14
  priority classes 1-7
  priority levels 1-7
thunk 2-19, 3-12
thunk layer 1-2
thunk procedure 3-31
thunking layer 3-13
tiled local descriptor table (LDT) 3-16
tiled memory, maximum address of 3-16
tiling 3-16
time critical threads 1-7
time slice, minimum value 1-6
time slicing 1-6
time-critical threads 1-7
timer 4-28, 6-15
TITLE 8-32
title-bar icon 1-17
transformation 1-27
transformations 4-52
Translation Lookaside Buffer (TLB) 2-7
trap gates 2-10
type checking 2-8

# U

underscore macro, SOM 7-16
unnamed pipes 1-10
unnamed shared memory 4-8
USAGE_MEMORY 8-30
USAGE_OPENVIEW 8-30
USAGE_RECORD 8-30
user environments
  application-oriented 8-1
  object-oriented 8-1
user interaction with objects 8-5
user interface 1-16
user interface, object-oriented
  icons 8-2
  objects 8-2
  shadow of an object 8-2
  views of an object 8-2
user-generated data exchange 1-22

# V

value set window control 4-48
  messages
    VM_QUERYITEM 4-49
    VM_QUERYITEMATTR 4-49
    VM_QUERYMETRICS 4-49
    VM_QUERYSELECTEDITEM 4-49
    VM_SELECTITEM 4-49
    VM_SETITEM 4-49
    VM_SETITEMATTR 4-49
    VM_SETMETRICS 4-49
    VN_DRAGLEAVE 4-48
    VN_DRAGOVER 4-48
    VN_DROP 4-48
    VN_DROPHELP 4-48
    VN_ENTER 4-48
    VN_INITDRAG 4-48
    VN_KILLFOCUS 4-48
    VN_SELECT 4-48
    VN_SETFOCUS 4-48
  styles
    VS_BITMAP 4-48
    VS_BORDER 4-48
    VS_COLORINDEX 4-48
    VS_ICON 4-48
    VS_ITEMBORDER 4-48
    VS_RGB 4-48
    VS_RIGHTTOLEFT 4-48
    VS_TEXT 4-48
VCDROM 6-15
VDD
  BIOS 6-15
  CMOS/Real Time Clock 6-15
  COM 6-15
  customizing settings 6-5
  Disk/Diskette 6-15
  DMA Controller 6-15
  Expanded Memory Specification 6-15
  Extended Memory Specification 6-15
  hooking system traps 6-13
  Keyboard 6-15
  loading at boot process 6-14
  Mouse 6-15
  numeric coprocessor (80387) 6-15
  Printer 6-15
  Programmable Interrupt Controller 6-15
  routing IN/OUT instruction traps 6-2
  Timer 6-15
  VCDROM 6-15
  VDPMI 6-15
  VDPX 6-15
  Video (EGA, VGA, and 8514/A) 6-15
VDD Services 4-37
VDHPopUp 6-13
VDPMI 6-15
VDPMI.SYS 6-12

VDPX 6-15
VDPX.SYS 6-13
VEMM.SYS 6-7
Video (EGA, VGA, and 8514/A) 6-15
VIEWBUTTON 8-34
VIEWITEM 8-30
views of an object
  composed view 8-2
  contents view 8-2
  help view of an object 8-3
  settings view 8-3
VIO applications 4-35
Virtual Device Driver Model 6-14
Virtual Device Driver (VDD) 6-2
  definition of 6-2
Virtual device drivers
  virtual expanded memory manager 6-7
  virtual extended memory manager 6-9
Virtual Device Helper functions 6-2
Virtual Device Helper Services 6-13
virtual instance 6-4
virtual machine 2-11
virtual memory 4-1
virtual programmable interrupt controller 6-14
virtual registers 2-11
virtual screen 4-35
virtual 8086 mode 2-11, 3-4, 6-1, 6-4
  enabling 6-4
virtual 8086 mode process 6-2
virtual 8086 task 2-11, 2-12
virtual 8086 task, execution of 2-12
virtual 8086 (V86) task 6-1
visual representation of objects 8-5
VM_QUERYITEM 4-49
VM_QUERYITEMATTR 4-49
VM_QUERYMETRICS 4-49
VM_QUERYSELECTEDITEM 4-49
VM_SELECTITEM 4-49
VM_SETITEM 4-49
VM_SETITEMATTR 4-49
VM_SETMETRICS 4-49
VN_DRAGLEAVE 4-48
VN_DRAGOVER 4-48
VN_DROP 4-48
VN_DROPHELP 4-48
VN_ENTER 4-48
VN_INITDRAG 4-48
VN_KILLFOCUS 4-48
VN_SELECT 4-48
VN_SETFOCUS 4-48
VS_BITMAP 4-48
VS_BORDER 4-48
VS_COLORINDEX 4-48
VS_ICON 4-48
VS_ITEMBORDER 4-48
VS_RGB 4-48
VS_RIGHTTOLEFT 4-48
VS_TEXT 4-48
VXMS.SYS 6-9
V86 mode process 6-1

# W

WC_CONTAINER 4-46
WC_NOTEBOOK 4-44
WC_SLIDER 4-49
WC_VALUESET 4-48
WinAddProgram 4-37
WinAllocMem 4-37
WinAssociateHelpInstance 1-24
WinAvailMem 4-37
WinCheckButton 4-53
WinCheckMenuItem 4-53
WinCreateGroup 4-37
WinCreateHeap 4-37
WinCreateHelpInstance 1-24
WinCreateMsgQueue 3-4
WinCreateObject 4-40, 8-34, 8-40, 8-41, 8-44, 8-45
WinDefDlgProc 1-24
WinDefWindowProc 1-24
WinDeleteLboxItem 4-53
WinDeregisterObjectClass 4-40, 8-40, 8-44, 8-45
WinDestroyHeap 4-37
WinDestroyHelpInstance 1-24
WinDestroyObject 4-40, 8-41
window 1-3
window border 1-17
window class 1-15
window controls
  container 4-45
  notebook 4-44
  slider 4-49
  value set 4-48
window environment 1-13
window handle 1-15
window hierarchy 1-13
window procedure 1-15
window relationships 1-13
window sizing 1-15
window sizing buttons 1-17
window templates 1-21
window title 1-17
window view 1-17
window words area 1-15
window-manager functions 1-30
windowable application 3-2
windowable applications 3-2
windows 1-13
window, definition of 1-13
WinEnableControl 4-53
WinEnableMenuItem 4-53
WinEnumObjectClasses 4-40, 8-40, 8-45
WinFileDgl 4-43
WinFontDgl 4-43

WinFreeFileIcon 4-40
WinFreeIcon 8-40
WinFreeMem 4-37
WinInitialize 3-4
WinInsertLboxItem 4-53
WinInstStartApp 4-37
WinIsControlEnabled 4-53
WinIsMenuItemChecked 4-53
WinIsMenuItemEnabled 4-53
WinIsMenuItemValid 4-53
WinLoadFileIcon 4-40, 8-40
WinLockHeap 4-37
WinLockWindow 4-38
WinLockWindowUpdate 4-38
WinPopupMenu 4-51
WinPostMsg() function 3-31
WinQueryButtonCheckState 4-53
WinQueryClassThunkProc 4-42
WinQueryDefinition 4-37
WinQueryDesktopBkgnd 4-51
WinQueryLboxCount 4-53
WinQueryLboxItemText 4-53
WinQueryLboxItemTextLength 4-53
WinQueryLboxSelectedItem 4-53
WinQueryObject 8-35, 8-40
WinQueryProfileData 4-38
WinQueryProfileInt 4-38
WinQueryProfileSize 4-38
WinQueryProfileString 4-38
WinQueryProgramTitles 4-37
WinQueryWindowLockCount 4-38
WinQueryWindowMode 4-42
WinQueryWindowThunkProc 4-42
WinReallocMem 4-37
WinRegisterObjectClass 4-40, 8-23, 8-40, 8-44, 8-45
WinRegisterWinDestroy 4-38
WinReplaceObjectClass 4-40, 8-40, 8-45
WinRestoreWindowPos 4-40, 8-40
WinSendMsg() function 3-31
WinSetClassThunkProc 4-42
WinSetDesktopBkgnd 4-51
WinSetFileIcon 4-40, 8-40
WinSetLboxItemText 4-53
WinSetMenuItemText 4-53
WinSetObjectData 4-40, 8-34, 8-35, 8-40, 8-41
WinSetWindowThunkProc 4-42
WinShutdownSystem 4-40, 8-40
WinStoreWindowPos 4-40, 8-40
WinTerminateApp 4-37
WinWriteProfileData 4-38
WinWriteProfileString 4-38
WMP_MSG1 3-32
WMP_MSG2 3-32
WM_QUIT 3-4
Workplace 4-39
Workplace API
Workplace API Functions
  WinCreateObject 8-40
Workplace API Functions *(continued)*
  WinDeregisterObjectClass 8-40
  WinEnumObjectClasses 8-40
  WinFreeIcon 8-40
  WinLoadFileIcon 8-40
  WinQueryObject 8-40
  WinReplaceObjectClass 8-40
  WinRestoreWindowPos 8-40
  WinSetFileIcon 8-40
  WinSetObjectData 8-40
  WinShutdownSystem 8-40
  WinStoreWindowPos 8-40
Workplace Class List Object 8-45
workplace classes, designing 8-9
Workplace Programming Interface 8-1, 8-8
Workplace Shell 8-6
Workplace Storage Classes
  WPAbstract 8-9
  WPFileSystem 8-9
  WPTransient 8-9
WPAbstract class, Workplace 8-9
wpAddObjectGeneralPage, settings notebook method 8-11
wpAddSettingsPage 8-38
wpAddSettingsPages, settings notebook method 8-11
wpAddToObjectUseList 8-30
wpAddToObjUseList 8-31
wpAllocMem 8-31
wpClose 8-20, 8-31
wpclsFindObjectEnd 8-36, 8-37
wpclsFindObjectFirst 8-36, 8-37
wpclsFindObjectNext 8-36, 8-37
wpclsQueryDefaultHelp 8-36
wpclsQueryDefaultView 8-36
wpclsQueryDetails 8-36
wpclsQueryDetailsInfo 8-23, 8-36
wpclsQueryIcon 8-36
wpclsQueryIconData 8-36
wpclsQueryInstanceFilter 8-47
wpclsQueryInstanceType 8-47
wpclsQueryObject 8-36
wpclsQueryStyle 8-36
wpclsQueryTitle 8-36
wpCnrInsertObject 8-31
wpCnrRemoveObject 8-31
wpCopyObject 8-20
wpCreateFromTemplate 8-20
wpCreateShadowObject 8-20
wpDelete 8-20
wpDeleteFromObjUseList 8-30, 8-31
wpDisplayHelp 8-20, 8-38
WPFileSystem class, Workplace 8-9
wpFilterPopupMenu 8-13
wpFindUseItem 8-30, 8-31
wpFree 8-32
wpFreeMem 8-31
wpHide 8-20

wpInitData 8-32
wpInsertPopupMenuItem 8-38
wpInsertPopupMenuItems 8-13
wpInsertSettingsPage 8-38
wpInsertSettingsPage, settings notebook method 8-11
  General Page
  removing from settings notebook 8-12
  replacing in settings notebook 8-12
wpMenuItemHelpSelected 8-19, 8-38
wpMenuItemSelected 8-19, 8-38
wpModifyPopupMenu 8-13, 8-38
wpMoveObject 8-20
WPObject 8-8
WPObject Class Method Groups
  Direct Manipulation 8-9
  Error Handling 8-9
  Memory Management 8-9
  Object Usage 8-9
  Pop-up Menu 8-9
  Save/Restore State 8-9
  Settings-Notebook 8-9, 8-10
  Set/Cleanup 8-9
  Set/Query Object Information 8-9
WPObject Class Methods
  wpclsFindObjectEnd 8-36
  wpclsFindObjectFirst 8-36
  wpclsFindObjectNext 8-36
  wpclsQueryDefaultHelp 8-36
  wpclsQueryDefaultView 8-36
  wpclsQueryDetails 8-36
  wpclsQueryDetailsInfo 8-36
  wpclsQueryIcon 8-36
  wpclsQueryIconData 8-36
  wpclsQueryObject 8-36
  wpclsQueryStyle 8-36
  wpclsQueryTitle 8-36
WPObject KEYNAMES
  CONCURRENTVIEW 8-34
  HELPPANEL 8-32
  ICONFILE 8-32
  ICONPOS 8-33
  ICONRESOURCE 8-33
  MINWIN 8-34
  NOCOPY 8-33
  NODELETE 8-32
  NODRAG 8-34
  NOMOVE 8-33
  NOPRINT 8-33
  NORENAME 8-34
  NOSHADOW 8-33
  NOTVISIBLE 8-33
  OBJECTID 8-33
  OPEN 8-34
  TEMPLATE 8-32
  TITLE 8-32
  VIEWBUTTON 8-34
wpOpen 8-20, 8-31, 8-38
wpPrintObject 8-20, 8-46
wpQueryDetailsData 8-23
wpRegisterView 8-38
wpRestore 8-20
wpSaveDeferred 8-29
wpSaveImmediate 8-29
wpSave/wpRestoreData 8-29
wpSave/wpRestoreLong 8-29
wpSave/wpRestoreState 8-29
wpSave/wpRestoreString 8-29
wpScanSetupString 8-32
wpSetup 8-32, 8-34, 8-41
wpSet/wpQueryDefaultHelp 8-22
wpSet/wpQueryDefaultView 8-22
wpSet/wpQueryIcon 8-22
wpSet/wpQueryIconData 8-22
wpSet/wpQueryStyle 8-22
wpSet/wpQueryTitle 8-22
WPTransient class, Workplace 8-9
wpUnInitData 8-32
writing files asynchronously 4-34
WYSIWYG 3-4

## X

XMS Version 2.0
  installation 6-9
  restrictions 6-9
  Virtual Extended Memory Manager (VXMS) 6-9
  VXMS.SYS 6-9

## Z

z order 1-14

# Numerics

0:32 memory model 2-13
16-Bit Functions
  Code-Page Management
    DosCaseMap 4-34
    DosGetCollate 4-34
    DosGetCp 4-34
    DosGetCtryInfo 4-34
    DosGetDBCSEv 4-34
    DosSetCp 4-34
    DosSetProcCp 4-34
  Controlling Threads
    DosResumeThread 4-13
    DosSuspendThread 4-13
  Creating Threads
    DosCreateThread 4-12
  Debugging Programs
    DosPtrace 4-16
  Determining Available Memory
    DosMemAvail 4-11
  Device I/O
    DosBeep 4-30
    DosCLIAccess 4-30
    DosDevConfig 4-30

16-Bit Functions *(continued)*
Device I/O *(continued)*
DosDevIOCtl 4-30
DosDevIOCtl2 4-30
DosPhysicalDisk 4-30
DosPortAccess 4-30
Discarding Memory Objects
DosLockSeg 4-11
DosUnlockSeg 4-11
DosGetShrSeg 4-2
Dynamic Linking
BadDynLink 4-29
DosFreeModule 4-29
DosFreeResource 4-29
DosGetEnv 4-29
DosGetMachineMode 4-29
DosGetModeName 4-29
DosGetModHandle 4-29
DosGetProcAddr 4-29
DosGetResource2 4-29
DosGetVersion 4-29
DosLoadModule 4-29
DosQAppType 4-29
Ending Other Process
DosKillProcess 4-14
Error Management
DosErrClass 4-35
DosError 4-35
Exiting from Threads and Processes
DosExit 4-13
File Initialization
WinQueryProfileData 4-38
WinQueryProfileInt 4-38
WinQueryProfileSize 4-38
WinQueryProfileString 4-38
WinWriteProfileData 4-38
WinWriteProfileString 4-38
File Systems
DosBufReset 4-31
DosChDir 4-31
DosChgFilePtr 4-31
DosClose 4-31
DosCopy 4-31
DosDelete 4-31
DosDupHandle 4-31
DosEditName 4-31
DosEnumAttribute 4-31
DosFileIO 4-31
DosFileLocks 4-31
DosFindClose 4-31
DosFindFirst 4-31
DosFindNext 4-31
DosFindNotifyClose 4-31
DosFindNotifyFirst 4-31
DosFindNotifyNext 4-31
DosFSAttach 4-31
DosFSCtl 4-31
DosMkDir 4-31
DosMove 4-31

16-Bit Functions *(continued)*
File Systems *(continued)*
DosNewSize 4-31
DosOpen 4-31
DosQCurDir 4-31
DosQCurDisk 4-31
DosQFHandState 4-31
DosQFileInfo 4-31
DosQFSAttach 4-31
DosQFSInfo 4-31
DosQHandType 4-31
DosQPathInfo 4-31
DosQSysInfo 4-31
DosQueryFileMode 4-31
DosQVerify 4-31
DosRead 4-31
DosReadAsync 4-31
DosRmDir 4-31
DosScanEnv 4-31
DosSearchPath 4-31
DosSelectDisk 4-31
DosSetFHandState 4-32
DosSetFileInfo 4-32
DosSetFileMode 4-32
DosSetFSInfo 4-32
DosSetMaxFH 4-32
DosSetPathInfo 4-32
DosSetVerify 4-32
DosShutDown 4-32
DosWrite 4-32
DosWriteAsync 4-32
Freeing Memory
DosFreeMem 4-4
Generating Dynamic Code
DosCreateCSAlias 4-9
Getting Thread and Process Information
DosGetEnv 4-14
DosGetInfoSeg 4-14
DosGetPID 4-14
DosGetPPID 4-14
DosGetPrty 4-14
Handling Critical Sections
DosEnterCritSec 4-14
DosExitCritSec 4-14
Heap Management
WinAllocMem 4-37
WinAvailMem 4-37
WinCreateHeap 4-37
WinDestroyHeap 4-37
WinFreeMem 4-37
WinLockHeap 4-37
WinReallocMem 4-37
Installed Program List
PrfAddProgram 4-37
PrfChangeProgram 4-37
PrfCreateGroup 4-37
PrfDestroyGroup 4-37
PrfQueryDefinition 4-37
PrfQueryProgramCategory 4-37

16-Bit Functions *(continued)*
Installed Program List *(continued)*
PrfQueryProgramHandle 4-37
PrfQueryProgramTitles 4-37
PrfRemoveProgram 4-37
WinAddProgram 4-37
WinCreateGroup 4-37
WinInstStartApp 4-37
WinQueryDefinition 4-37
WinQueryProgramTitles 4-37
WinTerminateApp 4-37
Memory Allocation
DosAllocHuge 4-2
DosAllocSeg 4-2
Memory Suballoction
DosSubAllocMem 4-4
DosSubFreeMem 4-4
DosSubSetMem 4-4
Memory-Management
DosAllocHuge 4-2
DosAllocSeg 4-2
DosAllocShrSeg 4-2
DosCreateCSAlias 4-2
DosGetHugeShift 4-2
DosGetSeg 4-2
DosGiveSeg 4-2
DosLockSeg 4-2
DosMemAvail 4-2
DosReallocHuge 4-2
DosReallocSeg 4-2
DosSizeSeg 4-2
DosSubAllocMem 4-2
DosSubFreeMem 4-2
DosSubSetMem 4-2
DosUnlockSeg 4-2
Memory-Management Functions
DosFreeSeg 4-2
Message Retrieval
DosGetMessage 4-34
DosInsMessage 4-34
DosPutMessage 4-34
Pop-Up Menus
WinPopUpMenu 4-51
Querying File Mode
DosQFileMode 4-33
Querying System Information
DosQSysInfo 4-33
Reading Asynchronously
DosReadAsync 4-33
Searching Directories
DosFindClose 4-32
DosFindFirst 4-32
DosFindNext 4-32
Semaphores
DosCloseSem 4-19
DosCreateSem 4-19
DosFSRamSemClear 4-19
DosFSRamSemRequest 4-19
DosMuxSemWait 4-19

16-Bit Functions *(continued)*
Semaphores *(continued)*
DosOpenSem 4-19
DosSemClear 4-19
DosSemRequest 4-19
DosSemSet 4-19
DosSemSetWait 4-19
DosSemWait 4-19
Session Management
DosSelectSession 4-35
DosSetSession 4-35
DosSMRegisterDD 4-35
DosStartSession 4-35
DosStopSession 4-35
Setting Available Number of File Handles
DosSetMaxFH 4-34
Setting the File Mode
DosSetFileMode 4-33
Signals
DosHoldSignal 4-35
DosSetSigHandler 4-35
Spooler
DosPrintDestAdd 4-39
DosPrintDestControl 4-39
DosPrintDestDel 4-39
DosPrintDestEnum 4-39
DosPrintDestGetInfo 4-39
DosPrintDestSetInfo 4-39
DosPrintDriverEnum 4-39
DosPrintJobContinue 4-39
DosPrintJobDel 4-39
DosPrintJobEnum 4-39
DosPrintJobGetId 4-39
DosPrintJobGetInfo 4-39
DosPrintJobPause 4-39
DosPrintJobSetInfo 4-39
DosPrintPortEnum 4-39
DosPrintQAdd 4-39
DosPrintQContinue 4-39
DosPrintQDel 4-39
DosPrintQEnum 4-39
DosPrintQGetInfo 4-39
DosPrintQPause 4-39
DosPrintQProcessorEnum 4-39
DosPrintQPurge 4-39
DosPrintQSetInfo 4-39
Starting Programs
DosExecPgm 4-16
Using Named Pipes
DosCallNmPipe 4-27
DosConnectNmPipe 4-27
DosCreateNmPipe 4-27
DosDisConnectNmPipe 4-27
DosPeekNmPipe 4-27
DosQNmPHandState 4-27
DosQueryNmPipeInfo 4-27
DosQueryNmPipeSemState 4-27
DosRawReadNmPipe 4-27
DosRawWriteNmPipe 4-27

16-Bit Functions *(continued)*
Using Named Pipes *(continued)*
DosSetNmHandInfo 4-27
DosSetNmPipeSem 4-27
DosTransactNmPipe 4-27
DosWaitNmPipe 4-27
Using Named Shared Memory
DosAllocShrSeg 4-6
DosGetShrSeg 4-6
Using Queues
DosCloseQueue 4-28
DosCreateQueue 4-28
DosOpenQueue 4-28
DosPeekQueue 4-28
DosPurgeQueue 4-28
DosQueryQueue 4-28
DosReadQueue 4-28
DosWriteQueue 4-28
Using Threads and Processes
DosCallBack 4-12
DosCreateThread 4-12
DosCWait 4-12
DosEnterCritSec 4-12
DosExecPgm 4-12
DosExit 4-12
DosExitCritSec 4-12
DosExitList 4-12
DosGetEnv 4-12
DosGetInfoSeg 4-12
DosGetPID 4-12
DosGetPPID 4-12
DosGetPrty 4-12
DosKillProcess 4-12
DosPTrace 4-12
DosQSysInfo 4-12
DosResumeThread 4-12
DosRetForward 4-12
DosR2StackRealloc 4-12
DosSetPrty 4-12
DosSuspendThread 4-12
Using Timers
DosGetDateTime 4-28
DosSetDateTime 4-28
DosSleep 4-28
DosTimerAsync 4-28
DosTimerStart 4-28
DosTimerStop 4-28
Using Unnamed Pipes
DosCreatePipe 4-26
DosMakePipe 4-26
Using Unnamed Shared Memory
DosGetSeg 4-8
DosGiveSeg 4-8
Waiting for Threads
DosCWait 4-14
Window Locking
WinLockWindow 4-38
WinLockWindowUpdate 4-38
WinQueryWindowLockCount 4-38
16-Bit Functions *(continued)*
Window Management
WinRegisterWinDestroy 4-38
Writing Asynchronously
DosWriteAsync 4-34
16-bit semaphores, shortcomings 4-17
16-bit subsystems 4-37
16:16 aliases 2-19
16→32 thunk 3-14
32-Bit Functions
Bit Maps
GpiDrawBits 4-51
Characters
GpiQueryCharBreakExtra 4-52
GpiQueryCharExtra 4-52
GpiQueryCharOutline 4-52
GpiQueryFaceString 4-52
GpiSetCharBreakExtra 4-52
GpiSetCharExtra 4-52
Checking a Process's Virtual-Memory Map
DosQueryMem 4-11
Code-Page Management
DosMapCase 4-34
DosQueryCollate 4-34
DosQueryCp 4-34
DosQueryCtryInfo 4-34
DosQueryDBCSEnv 4-34
DosSetProcessCp 4-34
Controlling Threads
DosResumeThread 4-13
DosSuspendThread 4-13
Creating Threads
DosCreateThread 4-12
Customizing Help
DdfBeginList 4-41
DdfBitmap 4-41
DdfEndList 4-41
DdfHyperText 4-41
DdfInform 4-41
DdfInitialize 4-41
DdfListItem 4-41
DdfMetafile 4-41
DdfPara 4-41
DdfSetColor 4-41
DdfSetFont 4-41
DdfSetFontStyle 4-41
DdfSetFormat 4-41
DdfSetTextAlign 4-41
DdfText 4-41
Debugging Programs
DosDebug 4-16
Desktop Background
WinQueryDesktopBkgnd 4-51
WinSetDesktopBkgnd 4-51
Device I/O
DosBeep 4-30
DosDevConfig 4-30
DosDevIOCtl 4-30
DosPhysicalDisk 4-30

32-Bit Functions *(continued)*
Dialog Boxes
WinFileDgl 4-43
WinFontDgl 4-43
DosAllocMem 4-2
DosSetFHState 4-32
Dynamic Linking
DosFreeModule 4-29
DosFreeResource 4-29
DosGetResource 4-29
DosLoadModule 4-29
DosQueryAppType 4-29
DosQueryModuleHandle 4-29
DosQueryModuleName 4-29
DosQueryProcAddr 4-29
DosQueryProcType 4-29
DosQueryResourceSize 4-29
Ending Other Process
DosKillProcess 4-14
Error Management
DosErrClass 4-35
DosError 4-35
Exception Management
DosAcknowledgeSignalException 4-36
DosEnterMustComplete 4-36
DosExitMustComplete 4-36
DosRaiseException 4-36
DosSendSignalException 4-36
DosSetExceptionHandler 4-36
DosSetSignalExceptionFocus 4-36
DosUnsetExceptionHandler 4-36
DosUnwindException 4-36
Exiting from Threads and Processes
DosExit 4-13
File Systems
DosCancelLockRequest 4-32
DosClose 4-31
DosCopy 4-31
DosCreateDir 4-31
DosDelete 4-31
DosDeleteDir 4-31
DosDupHandle 4-31
DosEditName 4-31
DosEnumAttribute 4-31
DosFindClose 4-31
DosFindFirst 4-31
DosFindNext 4-31
DosFSAttach 4-31
DosFSCtl 4-31
DosMove 4-31
DosOpen 4-31
DosQueryCurrentDir 4-31
DosQueryCurrentDisk 4-31
DosQueryFHState 4-31
DosQueryFileInfo 4-31
DosQueryFSAttach 4-31
DosQueryFSInfo 4-31
DosQueryHType 4-31
DosQueryPathInfo 4-31

32-Bit Functions *(continued)*
File Systems *(continued)*
DosQuerySysInfo 4-31
DosQueryVerify 4-31
DosRead 4-31
DosResetBuffer 4-31
DosScanEnv 4-31
DosSearchPath 4-31
DosSetCurrentDir 4-31
DosSetDefaultDisk 4-31
DosSetFileInfo 4-32
DosSetFileLocks 4-31
DosSetFilePtr 4-31
DosSetFileSize 4-31
DosSetFSInfo 4-32
DosSetMaxFH 4-32
DosSetPathInfo 4-32
DosSetRelMaxFH 4-32
DosSetVerify 4-32
DosShutDown 4-32
DosWrite 4-32
Fonts
GpiLoadPublicFonts 4-52
GpiQueryLogicalFont 4-52
GpiUnloadPublicFonts 4-52
Freeing Memory
DosFreeMem 4-4
Generating Dynamic Code
DosAllocMem 4-9
Getting Thread and Process Information
DosGetInfoBlocks 4-14
DosQuerySysInfo 4-14
Handling Critical Sections
DosEnterCritSec 4-14
DosExitCritSec 4-14
Memory Allocation
DosAllocMem 4-2
Memory Suballocation
DosSubUnsetMem 4-4
Memory-Management
DosAllocSharedMem 4-2
DosFreeMem 4-2
DosGetNamedSharedMem 4-2
DosGetSharedMem 4-2
DosGiveSharedMem 4-2
DosQueryMem 4-2
DosSetMem 4-2
DosSubSetMem 4-2
DosSubUnsetMem 4-2
Memory-Management Functions
DosSubAllocMem 4-2
DosSubFreeMem 4-2
Message Retrieval
DosInsertMessage 4-34
DosPutMessage 4-34
DosQueryMessageCp 4-34
Migration
WinQueryClassThunkProc 4-42
WinQueryWindowMode 4-42

32-Bit Functions *(continued)*
Migration *(continued)*
WinQueryWindowThunkProc 4-42
WinSetClassThunkProc 4-42
WinSetWindowThunkProc 4-42
Paths and Regions
GpiPathToRegion 4-51
Polylines
GpiPolylineDisjoint 4-52
Querying File Mode
DosQueryFileInfo 4-33
Querying System Information
DosQuerySysInfo 4-33
Regions
GpiFloodFill 4-51
GpiFrameRegion 4-51
Searching Directories
DosFindClose 4-32
DosFindFirst 4-32
DosFindNext 4-32
Semaphores
DosAddMuxWaitSem 4-19
DosCloseEventSem 4-19
DosCloseMutexSem 4-19
DosCloseMuxWaitSem 4-19
DosCreateEventSem 4-19
DosCreateMutexSem 4-19
DosCreateMuxWaitSem 4-19
DosDeleteMuxWaitSem 4-19
DosOpenEventSem 4-19
DosOpenMutexSem 4-19
DosOpenMuxWaitSem 4-19
DosPostEventSem 4-19
DosQueryEventSem 4-19
DosQueryMutexSem 4-19
DosQueryMuxWaitSem 4-19
DosReleaseMutexSem 4-19
DosRequestMutexSem 4-19
DosResetEventSem 4-19
DosWaitEventSem 4-19
DosWaitMuxWaitSem 4-19
Session Management
DosSelectSession 4-35
DosSetSession 4-35
DosStartSession 4-35
DosStopSession 4-35
Setting Available Number of File Handles
DosSetMaxFH 4-34
DosSetRelMaxFH 4-34
Setting Memory Commitment and Access
DosSetMem 4-11
Signalling Events with Semaphores
DosCloseEventSem 4-20
DosCreateEventSem 4-20
DosOpenEventSem 4-20
DosPostEventSem 4-20
DosQueryEventSem 4-20
DosResetEventSem 4-20
DosWaitEventSem 4-20

32-Bit Functions *(continued)*
Spooler
SplControlDevice 4-39
SplCreateDevice 4-39
SplCreateQueue 4-39
SplDeleteDevice 4-39
SplDeleteJob 4-39
SplDeleteQueue 4-39
SplEnumDevice 4-39
SplEnumDriver 4-39
SplEnumJob 4-39
SplEnumPort 4-39
SplEnumQueue 4-39
SplEnumQueueProcessor 4-39
SplHoldJob 4-39
SplHoldQueue 4-39
SplPurgeQueue 4-39
SplQueryDevice 4-39
SplQueryJob 4-39
SplQueryJobId 4-39
SplQueryQueue 4-39
SplReleaseJob 4-39
SplReleaseQueue 4-39
SplSetDevice 4-39
SplSetJobInfo 4-39
SplSetQueue 4-39
Starting Programs
DosExecPgm 4-16
Transformations
GpiConvertWithMatrix 4-52
Using Named Pipes
DosCallNPipe 4-27
DosConnectNPipe 4-27
DosCreateNPipe 4-27
DosDisConnectNPipe 4-27
DosPeekNPipe 4-27
DosQueryNPHState 4-27
DosQueryNPipeInfo 4-27
DosQueryNPipeSemState 4-27
DosSetNPHState 4-27
DosSetNPipeSem 4-27
DosTransactNPipe 4-27
DosWaitNPipe 4-27
Using Named Shared Memory
DosAllocSharedMem 4-6
DosGetNamedSharedMem 4-6
Using Queues
DosCloseQueue 4-28
DosCreateQueue 4-28
DosOpenQueue 4-28
DosPeekQueue 4-28
DosPurgeQueue 4-28
DosQueryQueue 4-28
DosReadQueue 4-28
DosWriteQueue 4-28
Using Semaphores for Multiple Waiting
DosAddMuxWaitSem 4-25
DosCloseMuxWaitSem 4-25
DosCreateMuxWaitSem 4-25

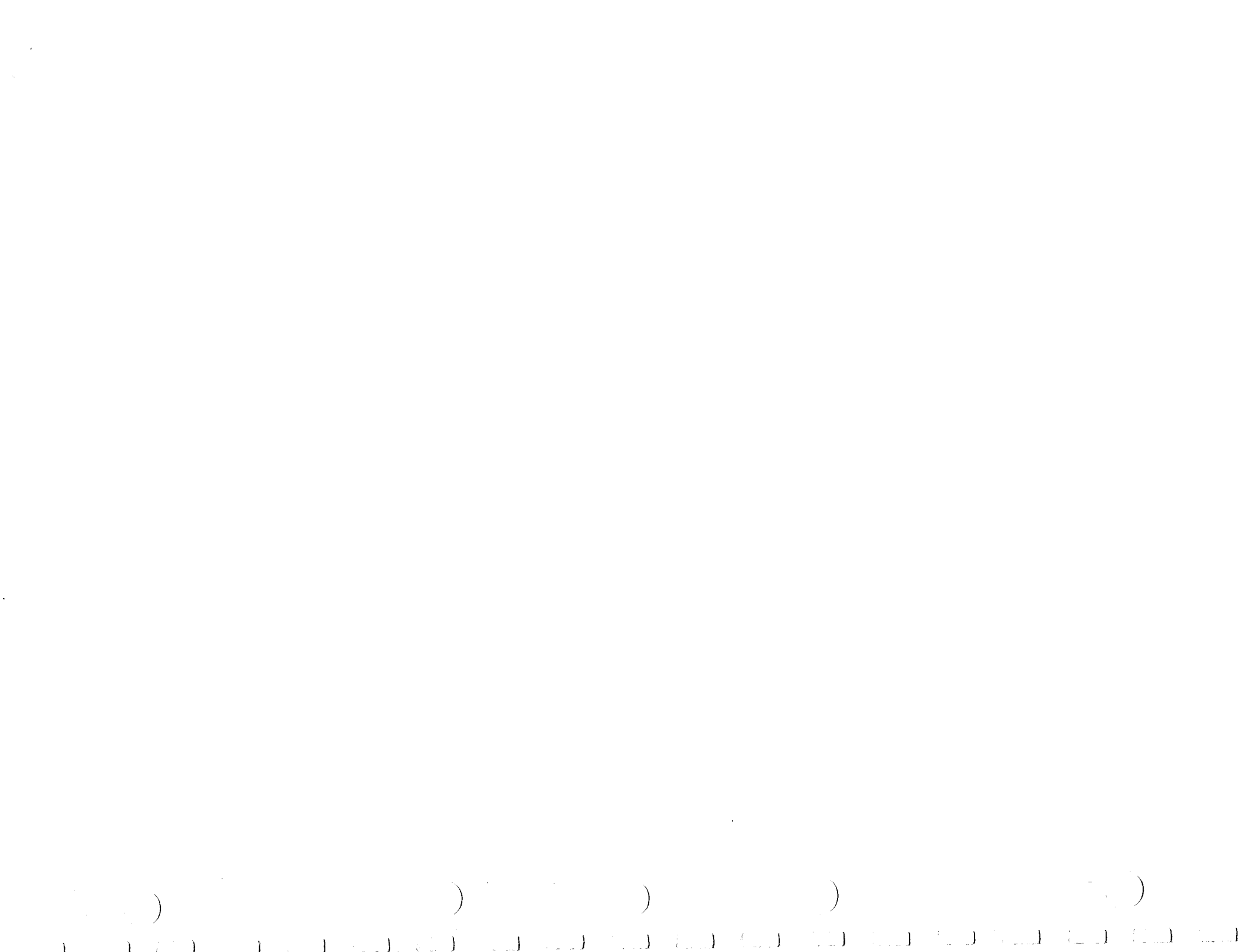
32-Bit Functions *(continued)*
Using Semaphores for Multiple Waiting *(continued)*
DosDeleteMuxWaitSem 4-25
DosOpenMuxWaitSem 4-25
DosQueryMuxWaitSem 4-25
DosWaitMuxWaitSem 4-25
Using Semaphores for Mutual Exclusion
DosCloseMutexSem 4-21
DosCreateMutexSem 4-21
DosOpenMutexSem 4-21
DosQueryMutexSem 4-21
DosReleaseMutexSem 4-21
DosRequestMutexSem 4-21
Using Threads and Processes
DosCreateThread 4-12
DosDebug 4-12
DosEnterCritSec 4-12
DosExecPgm 4-12
DosExit 4-12
DosExitCritSec 4-12
DosExitList 4-12
DosGetInfoBlocks 4-12
DosKillProcess 4-12
DosQSysInfo 4-12
DosResumeThread 4-12
DosSetPriority 4-12
DosSuspendThread 4-12
DosWaitChild 4-12
DosWaitThread 4-12
Using Timers
DosAsyncTimer 4-28
DosGetDateTime 4-28
DosSetDateTime 4-28
DosSleep 4-28
DosStartTimer 4-28
DosStopTimer 4-28
Using Unnamed Shared Memory
DosGetSharedMem 4-8
DosGiveSharedMem 4-8
VDD Services
DosCloseVDD 4-37
DosOpenVDD 4-37
DosRequestVDD 4-37
Waiting for Threads
DosWaitChild 4-14
DosWaitThread 4-14
Workplace
WinCreateObject 4-40
WinDeregisterObjectClass 4-40
WinDestroyObject 4-40
WinEnumObjectClasses 4-40
WinFreeFileIcon 4-40
WinLoadFileIcon 4-40
WinRegisterObjectClass 4-40
WinReplaceObjectClass 4-40
WinRestoreWindowPos 4-40
WinSetFileIcon 4-40
WinSetObjectData 4-40
WinShutdownSystem 4-40

32-Bit Functions *(continued)*
Workplace *(continued)*
WinStoreWindowPos 4-40
32-Bit Migration 4-41
32-Bit OS/2 Memory Layout 3-15
32-Bit PM Helper Macros
WinCheckButton 4-53
WinCheckMenuItem 4-53
WinDeleteLboxItem 4-53
WinEnableControl 4-53
WinEnableMenuItem 4-53
WinInsertLboxItem 4-53
WinIsControlEnabled 4-53
WinIsMenuItemChecked 4-53
WinIsMenuItemValid 4-53
WinMenuItemEnabled 4-53
WinQueryButtonCheckState 4-53
WinQueryLboxCount 4-53
WinQueryLboxItemText 4-53
WinQueryLboxItemTextLength 4-53
WinQueryLboxSelectedItem 4-53
WinSetLboxItemText 4-53
WinSetMenuItemText 4-53
32→16 thunk 3-13
64-bit barrel shifter 2-2
80286 processor, mode switching 6-3
80386 addressing modes
protect mode 2-3
real mode 2-3
80386 architecture 2-1, 2-2, 2-3, 2-6, 2-8, 2-9, 2-10, 2-11, 2-12, 2-13
coprocessing 2-13
input/output processing 2-11
interrupts 2-10
memory addressing 2-3
numeric coprocessor 2-12
paging 2-6
physical characteristics 2-2
protection 2-8, 2-9, 2-10
limit checking 2-8
privilege levels 2-9
procedure entry points, restrictions 2-9
reserved instructions 2-10
Type Checking 2-8
virtual 8086 mode 2-11
80386 chip
control registers
CR0 2-3
CR1 2-3
CR2 2-3
CR3 2-3
debug registers
DR0 2-3
DR1 2-3
DR2 2-3
DR3 2-3
DR4 2-3
DR5 2-3
DR6 2-3

80386 chip *(continued)*
debug registers *(continued)*
DR7 2-3
memory management registers
Global Descriptor Table Register (GDTR) 2-3
Interrupt Descriptor Table Register (IDTR) 2-3
Local Descriptor Table Register (LDTR) 2-3
Task Register (TR) 2-3
16-bit registers, number of 2-3
32-bit general registers, number of 2-2
80386 processor
dedicated units
bus interface unit 2-2
code prefetch unit 2-2
execution unit 2-2
instruction decode unit 2-2
paging unit 2-2
segmentation unit 2-2
80386 processor, mode switching 6-3
8086 emulation 6-2
8086 instruction decoding 6-2

# Special Characters

.ASSOCTABLE EAs, Workplace 8-46
.CS2, SOM C-language binding file 7-12
.C, SOM C-language binding file 7-12
.DEF, module-definition file 5-6
.DEF, SOM C-language binding file 7-12
.H, public header file 7-12, 7-23
.H, SOM C-language binding file 7-12
.IH, class implementation file 7-22
.IH, SOM C-language binding file 7-12
.LIB files 3-12
.LONGNAME EAs, Workplace 8-46
.PH, SOM C-language binding file 7-12
.PSC, SOM C-language binding file 7-12
.SC, SOM C-language binding file 7-12, 7-29
<WP_CONFIG> 8-41
<WP_DESKTOP> 8-41
<WP_DRIVE> 8-41
<WP_INFO> 8-41
<WP_NOWHERE> 8-41
<WP_START> 8-41
<WP_SYSTEM> 8-41
<WP_TEMPS> 8-41
/Gt+ compilation option 3-10
_Far16 keyword 3-29
_Far16 _Pascal keyword 3-11, 3-20
_Seg16 keyword 3-11, 3-20, 3-29, 3-30
:acviewport 4-41
:br 4-41
:ddf 4-41
:docprof 4-41
:font 4-41
:table 4-41
:title 4-41
#pragma linkage directive 3-29
#pragma seg16 directive 3-31
#pragma seg16 directive, use of 3-19
#pragma stack16 directive 3-29



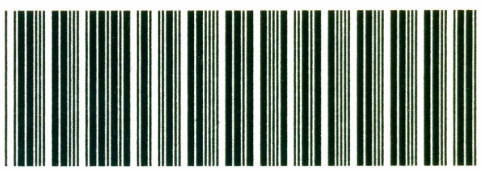


International Business
Machines Corporation

Printed in the
United States of America



10G6260



S10G-6260-00

P10G6260